THE DESIGN, IMPLEMENTATION AND ANALYSIS OF A LOW COST FLIGHT SIMULATOR USING RMC'S 3DOF MOTION PLATFORM

LE DÉSIGN, L'IMPLÉMENTATION ET L'ANALYSE D'UN SIMULATEUR DE VOL À FAIBLE COÛT UTILISANT LA PLATFORME DE MOUVEMENT À TROIS DÉGRÉS DE LIBERTÉ DU CMR

A Thesis Submitted

to the Faculty of the Royal Military College of Canada

by

Jean Le Bouthillier, RMC

Second Lieutenant

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science.

October 2010

© This thesis may be used within the Department of National Defence but copyright for open publication remains the property of the author.

ROYAL MILITARY COLLEGE OF CANADA
COLLÈGE MILITAIRE ROYAL DU CANADA

DIVISION OF GRADUATE STUDIES AND RESEARCH
DIVISION DES ÉTUDES SUPÉRIEURES ET DE LA RECHERCHE

This is to certify that the thesis prepared by / Ceci certifie que la thèse rédigée par

Jean Le Bouthillier

entitled / intitulée

The design, implementation and analysis of a low cost flight simulator using RMC's 3DOF motion platform

Le désign, l'implémentation et l'analyse d'un simulateur de vol à faible cout utilisant la platforme de mouvement à trois degrés de liberté du CMR

complies with the Royal Military College of Canada regulations and that it meets the accepted standards of the Graduate School with respect to quality / satisfait aux règlements du Collège militaire royal du Canada et qu'elle respecte les normes acceptées par la Faculté des études supérieures quant à la qualité.

for the degree of / pour le diplôme de

MASTERS OF SCIENCE IN COMPUTER SCIENCE
MAÎTRISE ÈS SCIENCES EN INFORMATIQUE

Signed by the final examining committee: / Signé par les membres du comité examinateur de la soutenance de thèse :

_____ Chair / Président

_____ External Examiner / Examinateur externe

_____ Int/External Examiner / Examinateur interne / externe

_____ Internal Examiner / Examinateur interne

_____ Supervisors / Directeurs de thèse

Approved by the Head of Department : /
Approuvé par le Directeur du Département : _____ Date: _____

To the Librarian: This thesis is not to be regarded as classified. /
Au Bibliothécaire : Cette thèse n'est pas considérée comme à publication restreinte.

_____
Supervisors / Directeurs de thèse

ACKNOWLEDGEMENTS

First, I would like to thank my parents for the great qualities they haved instilled in me. Their unwavering support, guidance and encouragements throughout the years in all my endeavours have been invaluable. They instilled in me the values of hardwork and discipline which have always served me well. My success is but an extension of their remarkable parenthood and dedication.

I would like to express my deepest thanks to Dr. Yawei Liang and Dr. Paul Allard for giving me the opportunity to study the challenging problem of flight simulation. Their knowledge, guidance and critical outlook allowed me to structure and improve my work. Their patience, availability and commitment to education are commendable attributes which played an essential role in the success of my work.

Within the department, I would like to thank Mr. Ben Brownlee and Dr.Yong Kui Wang for their support in developing solutions to problems which arose during the study. Their knowledge and willingness to support me in unfamiliar areas contributed greatly to the quality of this work. I also have to express my thanks to Martin Baxter of the Electrical Engineering department for his support in machining and metal working.

Next, I would like to thank the volunteer pilots who gave their time and allowed this study to benefit from their precious experience and opinions. The RMC Flying Club and the Ontario Fun Flyers organizations were instrumental in providing volunteers for this effort. My

outreach for volunteer pilots was met with great enthusiasm by Mr. Tucker Densmore and Mr. Barry Smith who recruited and organized volunteers.

Finally, I would also like to extend special thanks to my house mates Dan, Gabriel and Alex from whom I received many encouragements and motivation throughout the year. To Sylvia, I must thank you for your understanding, patience and support even as I cultivated the habit of working very late each night. It is clear to me that your support and love allowed me to focus my energies and creativity into this work.

ABSTRACT

Flight simulation is extensively used by the Canadian Forces for the training of pilots. The flight simulators in use often feature 6 degrees of freedom motion platforms and exhibit high levels of realism. In this study, a low cost flight simulator was designed and implemented using a 3 degree of freedom motion platform with low cost, commercially available components. The objective of this work was to investigate the merits and shortfalls of such a system. This document describes the design, implementation and analysis of this system.

Central to this work was the classical washout filter (WF) algorithm, a widely adopted motion cueing algorithm used to generate the platform's motion. An inertial measurement unit was used to study and optimize the classical WF algorithm. Finally, the system was assessed by collecting pilot ratings through a pilot validation exercise. Results showed that most pilots were satisfied by the system and that the addition of motion to flight simulation greatly enhanced flight simulation. Pilots also identified areas for improvements and potential applications for the flight simulator developed in this study.

The empirical analysis of the system using an inertial measurement unit and the ratings obtained through a pilot validation exercise indicate that low cost 3 degrees of freedom flight simulator have potential for flight training.

RÉSUMÉ

La simulation du vol est couramment utilisée par les Forces Canadiennes pour l'entrainement des pilotes. Les simulateurs utilisés ont la plupart du temps des plateformes de mouvement avec 6 degrés de liberté et ont un niveau de réalisme très élevé. Dans ce mémoire, un simulateur de vol à faible coût fut developpé avec une plateforme de mouvement avec 3 degrés de liberté ainsi que des composantes disponibles commerciallement. Le but de cette étude fut d'investiguer les mérites et inconvénients d'un tel système. Ce document décrit le désign, l'implémentation et l'analyse de ce système.

L'algorithme classique de filtrage, un algorithme de mouvement très populaire, fut utilisé pour générer les mouvements de la plateforme. Un senseur de mouvement fut utilisé pour étudier et optimiser l'algorithme classique de filtrage. Finalement, les évaluations de pilotes furent receuillies par l'entremise d'un exercise de validation. Les résultats de cet exercise ont démontrés que l'addition de mouvement à la simulation du vol a un impact très positif. Les pilotes ont également formulés des suggestions pour améliorer le système et ils ont identifés des applications potentielles pour le simulateur de vol développé dans cette étude.

L'étude empirique du système avec un senseur de mouvement ainsi que les évaluations obtenues par l'entremise d'un exercise de validation indiquent que les simulateurs avec 3 degrés de liberté à faible coût ont un potentiel pour l'entrainement au vol.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS III

ABSTRACT V

RÉSUMÉ VI

TABLE OF CONTENTS VII

LIST OF FIGURES X

LIST OF TABLES XIV

LIST OF SYMBOLS XVI

CHAPTER 1 – INTRODUCTION 2

1.1 Introduction 2

1.2 Flight Simulation 2

1.2.1 Flight Simulation History 2
1.2.2 Advantages of Flight Simulation 3
1.2.3 Modern Flight Simulation 3

1.3 RMC 3DOF Motion Platform 4

1.4 Motivation 7

1.5 Aim 8

1.6 Contributions 8

1.7 Thesis Structure 10

CHAPTER 2 – MOTION DRIVE ALGORITHM AND 3DOF ADAPTATION 12

2.1 Introduction 12

2.2 Reference Frames and Axis System 13

2.3 Classical Washout Filter Algorithm 14

2.3.1 Translational Motion Channel 16
2.3.2 Rotational Motion Channel 17
2.3.3 Tilt-Coordination Channel 18
2.3.4 Scaling and Limiting 20
2.3.5 Filtering 21

2.4 Three Degree-of-Freedom Adaptation 23
2.4.1 Translational Motion Channel 23
2.4.2 Rotational Motion Channel 26
2.4.3 Tilt-Coordination Motion Channel 28
2.5 Summary 30
CHAPTER 3 – DESIGN 31
3.1 Introduction 31
3.2 Design Objectives 32
3.2.1 Achieving a Realistic Simulation 33
3.2.2 Cost Effectiveness 35
3.2.3 Software Portability 36
3.2.4 Ease of Operation 38
3.2.5 Maintainability 39
3.2.6 Safety 40
3.3 Constraints 42
3.3.1 Hardware Constraints 43
3.3.2 Software Constraints 46
3.4 Flight Simulator Components 47
3.4.1 Simulation Environment 49
3.4.2 Visual Display and Sound 51
3.4.3 Pilot Seat and Controls 54
3.4.4 Simulator Operation and Safety 61
3.4.5 Graphical User Interface 62
3.4.6 Programming Language 64
3.5 Summary 65
CHAPTER 4 – IMPLEMENTATION 66
4.1 Introduction 66
4.2 Software Implementation 67
4.2.1 Purpose and Use Cases 68
4.2.2 Functional Approach to Implementation 70
4.2.3 Mediator Design Pattern 71
4.2.4 Flight Simulator X Plug-in 73
4.2.5 Classical WF Algorithm Implementation 77
2.4.4 Calculating the Actuator Extensions, Velocities and Accelerations 81
4.2.6 Networking 83
4.2.7 Documentation 86
4.3 Software Validation 88
4.3.1 Technical Verification 88
4.3.2 User Validation 89
4.4 Hardware Implementation 90
4.4.1 Special Mounting Parts and Installation 91
4.5 Summary 94
CHAPTER 5 - EXPERIMENTAL STUDY AND PILOT VALIDATION 95

5.1 Introduction 95
5.2 Experimental Study of the System 95
5.2.1 Statement of the Problem and Objectives 96
5.2.2 Measures of Performance and Response Variables 97
5.2.3 Choice of Factors 100
5.2.4 Experimental Design 101
5.2.4 Experimental Procedure 106
5.3 Conduct of the Experiment 108
5.3.1 Reproducibility and Repeatability Considerations 108
5.3.2 Test Run #1 113
5.3.3 Solving the Problem of 2nd Order Filter Overshooting 119
5.3.4 Test Run #2 123
5.3.5 Test Run #3 130
5.3.5 Conclusion 132
5.4 Pilot Validation Exercise 133
5.4.1 Procedure 134
5.4.2 Pilot Ratings and Comments 136
5.5 Summary 140
CHAPTER 6 - CLOSING REMARKS 142
6.1 Introduction 142
6.2 Future Work 143
6.2.1 Classical WF Algorithm Improvement 143
6.2.2 Flight Simulator Hardware Improvement 144
6.2.3 Future Applications 146
ANNEXES 148
Annex A - Motion Platform Security Briefing 149
Annex B - Flight Simulator Components and Cost 151
Annex C - Flight Simulator Wiring Diagram 153
Annex D - UML Class Diagram 155
Annex E - Response Variable Calculation Code 157
Annex F - Crossbow IMU 320 Simulink Interface 160
Annex G - Recording and Playback 164
Annex H - Pilot Rating Questionnaire 168
Annex I - Pilot Information and Ratings 172
Annex J – Ethics Board Documentation 176
Annex K - C++ Implentation Code 188
REFERENCES 225

LIST OF FIGURES

Page

Figure 1 - RMC's Triad 1500-E 3DOF Motion Platform 5

Figure 2 - Motion Platform Actuator Assembly 5

Figure 3 - Reference Frames and Axis System 13

Figure 4 - Classical WF Algorithm [6] 15

Figure 5 - Example of Tilt-Coordination 19

Figure 6 - Scaling and Limiting [6] 20

Figure 7 - 3DOF Translational Motion Channel 24

Figure 8 - Translational *z* Component High-Pass Filter (ωn= 4 *rad/sec*) 25

Figure 9 – Translational Channel High-Pass Filter Sample Run (ωn= 4 rad/sec) 25

Figure 10 - 3DOF Rotational Motion Channel 27

Figure 11- Rotational Channel High-Pass Filter (ωn = 1 rad/sec) 27

Figure 12 - Rotational Channel High-Pass Filter Sample Run (ωn = 1 rad/sec) 28

Figure 13 – Tilt-Coordination Channel *x* Component Low-Pass Filter (ωn = 5 rad/sec) 29

Figure 14 - Tilt-Coordination Low-Pass Filter *x* Component Sample Run 29

Figure 15 - Objective Tree for RMC's 3DOF Flight Simulator 32

Figure 16 - Realistic Simulation Design Sub-Objectives 34

Figure 17 - Cost Effectiveness Design Sub-Objectives 36

Figure 18 - Software Portability Design Sub-Objectives 37

Figure 19 - Ease of Operation Design Sub-Objectives 39
Figure 20 - Maintainability Design Sub-Objectives 40
Figure 21 - Safety Design Sub-Objectives 41
Figure 22 - Design Constraints 43
Figure 23 - Design Hardware Constraints 43
Figure 24 - Serious Damage Sustained by RMC's 3DOF 45
Figure 25 - Hardware Collision Caused by a Design Flaw 45
Figure 26 - Design Software Constraints 47
Figure 27 - Interaction of Components in the Final Flight Simulator Design 48
Figure 28 - Z800 3D Visor 52
Figure 29 - TrackIR 5 with TrackIR PRO Clip 53
Figure 30 - Fatal1ty Sound Headset 54
Figure 31 - Corbeau A4 Racing Seat and Dimensions 55
Figure 32 - Corbeau Four Points Harness Belt 56
Figure 33- Logitech’s G940 Flight System 58
Figure 34 - Concept Drawing of the Pilot Seat and Controls 59
Figure 35 – Close-up on Control Adjustment Mechanisms Concept 60
Figure 36 - SquareD Heavy Duty 60A Safety Switch 62
Figure 37 - Final GUI Design 63
Figure 38 - General Interaction of Software Components and Flow of Information 67
Figure 39 - System Use Case Diagram 69
Figure 40 - Comparison between Coupled and Functional Approach 71
Figure 41 - The "Many to Many" Relationship and the Mediator Design Pattern 72

Figure 42 - SimConnect TCP/IP Communications Model 74

Figure 43 - Different Axis Systems between Classical WF Algorithm and FSX..... 77

Figure 44 - Simulink Implementation of the Classical WF Algorithm 79

Figure 45 - Internal Simulink Block View Sample 80

Figure 46 - Geometrical Layout of the Actuators (Top Frame View)..... 81

Figure 47 - Superimposition to Calculate Actuator Extensions..... 82

Figure 48 - Networking in the Flight Simulation System..... 83

Figure 49 - Classes Required for the Implementation of UDP..... 84

Figure 50 - Typical UDP Communication..... 85

Figure 51 - C++ Code Sample 87

Figure 52 - Installed Special Mounting Parts 92

Figure 53 - Head Mounted Display and TrackIR 5 93

Figure 54 - Comparison between Concept Draw and Actual Implementation..... 93

Figure 55 - The Position of the IMU 320 on RMC's 3DOF Motion Platform 100

Figure 56 - FSX's C-172 - Key Information and Geometrical References..... 102

Figure 57 – Plot of Climb Profile Motions against Time 104

Figure 58 - Plot of Standard Rate Turn Profile Motions against Time..... 105

Figure 59 - Plot of Descent Profile Motions against Time 106

Figure 60 – Noisy Crossbow Crossbow Angular Rate Signal..... 110

Figure 61 - Smoothing of a Typical Crossbow Angular Rate Signal 112

Figure 62 - Margin of Errors (Smoothed vs. Raw Crossbow Signals) 113

Figure 63 – *z* Axis Accelerations (Climb Profile vs Test Run #1) 116

Figure 64 - *y* Axis Accelerations (Turn Profile vs Test Run #1)..... 117

Figure 65 - *p* Angular Rates (Turn Profile vs Test Run #1) 119

Figure 66 – 2nd Order High Pass Filter Overshooting..... 120

Figure 67 - Overshoot Management Using a Derivative Switch..... 121

Figure 68 – Overshoot Management Using a Polarity Switch 121

Figure 69- The effect of taking the Derivative in the Tilt Limiting Block 124

Figure 70 - *z* Axis Accelerations (Climb Profile vs Test Run #2) 127

Figure 71 - *y* Axis Accelerations (Turn Profile vs Test Run #2) 128

Figure 72 - *p* Angular Rates (Turn Profile vs Test Run #1) 129

Figure 73 - Flight Simulator Wiring 154

Figure 74- Crossbow IMU 320 Packet Format..... 161

Figure 75 - Crossbow IMU 320 Simulink Interface 162

Figure 76 - Simulink Serial Configurations (IMU 320) 163

Figure 77 - Top Level View of Flight Recorder Block 165

Figure 78 - Inside View of Flight Recording Block 165

Figure 79 - Top Level View of Playback and Crossbow Blocks..... 166

Figure 80 - Inside View of Flight Playback block..... 167

LIST OF TABLES

Page

Table 1- RMC's 3DOF Motion Platform Limits 6

Table 2 - Filter Parameters with Most Favourable Pilot Response (UTIAS) [11] 22

Table 3 - Cost-Benefit Analysis of Flight Simulation Software 49

Table 4 - Cost-Benefit Analysis of the Three Major Types of Display 51

Table 5 - Cost-Benefit Analysis Between three Commercially Available Pilot Input Systems ... 57

Table 6 - Programming Languages Compatibility for Key Components 64

Table 7 - Information Requested from FSX Using FSXConnect 75

Table 8 - Port Allocation for UDP Traffic 86

Table 9 - Observations and Problems Discovered during User Validation 90

Table 10 - Special Mounting Parts 91

Table 11 - Crossbow IMU 320 Performance Data 100

Table 12 - Experimental Flight Profiles Manoeuvres 104

Table 13 - Experimental Procedure for Each Test Runs 107

Table 14 - Margin of Error for Three Identical Runs (Unsmoothed Signals) 109

Table 15 - Margin of Error for Three Identical Runs (Smoothed Signals) 112

Table 16 - Classical WF algorithm Configuration (Test Run #1) 114

Table 17 - Pilot Observations (Test Run #1) 115

Table 18 - Response Variables for Each Motion Channels (Test Run #1) 115

Table 19 - Classical WF algorithm Configuration (Test Run #2) 125

Table 20 - Pilot Observations (Test Run #2) 125

Table 21 - Response Variables for Each Motion Channels (Test Run #2) 126

Table 22 - Response Variable Variations between Tests Run #1 and #2 127

Table 23 - Classical WF algorithm Configuration (Test Run #3) 130

Table 24 - Pilot Observations (Test Run #3) 131

Table 25 - Response Variable Variations between Tests Run #2 and #3 132

Table 26 - Pilot Validation Exercise Procedure 134

Table 27 - Average Pilot Ratings for Flight Simulation Hardware 136

Table 28 – Summary of Pilot Complaints for the Flight Simulator Hardware 137

Table 29 – Average Pilot Ratings for Motion Simulation 138

Table 30 – Average Pilot Ratings for Motion Impact and Flight Training Potential 139

Table 31 - Pilot Suggestions for Potential Training Use 140

Table 32 - Pilot Information 173

Table 33 - Complete Hardware Ratings 174

Table 34 - Complete Motion Simulation Ratings 174

Table 35 - Complete Impact and Training Potential Ratings 175

LIST OF SYMBOLS

Acronyms:

3DOF	Three Degree-of-Freedom
6DOF	Six Degree-of-Freedom
C-172	Cessna 172 SP Skyhawk Airplane
CF	Canadian Forces
DLSE	Directorate of Land Synthetic Environment
DRDC	Defence Research and Development Canada
FAA	Federal Aviation Administration
FSX	Flight Simulator X
GUI	Graphical User Interface
IDE	Integrated Development Environment
IFR	Instrument Flight Rules
IMU	Inertial Measurement Unit
RMC	Royal Military College of Canada
SAE	Socitety of Automotive Engineers
SDK	Software Development Kit
SGN	Sign operation (taking only the sign of a variable)
SSSI	Sarnicola Simulation Systems Inc.
UML	Unified Modelling Language
USB	Universal Serial Bus
UTIAS	University of Toronto Institute for Aerospace Studies
VGA	Video graphics array
VS2008	Microsoft Visual Studio Professional 2008
WF	Washout Filter
Winsock	Windows Sockets

Variables:

α	Aircraft pitch angle
a_i	Individual actuator acceleration
$\overrightarrow{a_{S_I}}$	High pass filtered accelerations
$\overrightarrow{A_{S_I}}$	Actuator attachments position on the motion platform base
$\overrightarrow{B_{S_I}}$	Actuator attachments position on the upper motion platform frame
β	Aircraft bank angle
$\overrightarrow{\dot{\beta}_H}$	Euler angle rates of the rotational motion channel
$\overrightarrow{\dot{\beta}_{SH}}$	High-pass filtered Euler angle rates of the rotational motion channel
d_i	Individual actuator displacement
$\vec{S}$	Location of F_S with respect to F_I
$\overrightarrow{\beta_L}$	Low frequency tilt-coordination Euler angles
$\overrightarrow{\beta_S}$	Simulator Euler angles $[\phi, \theta, \psi]$
$\overrightarrow{\beta_{SH}}$	High frequency simulator Euler angles
$\overrightarrow{\beta_{SL}}$	Low frequency simulator Euler angles
$\overrightarrow{\beta_{SL}}$	Simulator low frequency Euler angles
$\vec{\epsilon}$	Angular acceleration
F_A	Aircraft body-axis reference frame
$\overrightarrow{f_{AA}}$	Aircraft accelerations $[a_x,\ a_y,\ a_z - g]$
F_B	Aircraft center of gravity reference frame
F_I	Earth-fixed inertial reference frame
$\overrightarrow{f_L}$	Low frequency aircraft accelerations
$\overrightarrow{f_1}$	Input of the tilt-coordination channel
$\overrightarrow{f_2}$	Intermediate earth referenced accelerations
F_{Pa}	Aircraft pilot's head reference frame
F_{Ps}	Simulator pilot's head reference frame
F_S	Simulator body-axis reference frame
$\overrightarrow{f_{SS}}$	Simulator accelerations $[a_x,\ a_y,\ a_z - g]$
$\vec{g}$	Acceleration due to gravity [*0, 0, 9.81 m·s*$^{-2}$]

$\overrightarrow{g_{xyz}}$	Aircraft acceleration due to gravity [g_x, g_y, g_z]
K_i	Scaling coefficient of the scaling and limiting process
L_i	Limiting value of the scaling and limiting process
l_i	Individual actuator length
L_{IS}	Euler rotation matrix transforming accelerations from F_A to F_I
$\overrightarrow{P_A}$	Location of F_A with respect to F_B
$\overrightarrow{R_A}$	Location of F_{Pa} with respect to F_A
$\overrightarrow{R_S}$	Location of F_{Ps} with respect to F_S
$\overrightarrow{RV_A}$	Response variables for accelerations
$\overrightarrow{RV_\omega}$	Response variables for angular rates
$\overrightarrow{S_I}$	Simulator translational displacement [d_x, d_y, d_z]
$\vec{S}$	Location of F_S with respect to F_I
S_j	Smoothed point of a dataset at index j
T_S	Transformation from angular velocity to Euler angle rates
v_i	Individual actuator velocity
$\overrightarrow{\omega_{AA}}$	Aircraft angular rates $[p, q, r]$
$\overrightarrow{\omega_{SS}}$	Simulator angular rates $[p, q, r]$
σ	Standard deviation
μ	Arithmetic mean
ζ	Filter damping ratio
ω_n	Filter break frequency

CHAPTER 1 – INTRODUCTION

1.1 Introduction

In this chapter, the topic of flight simulation from an historical and practical point of view is introduced to the reader. Then, the motion simulation hardware that makes this study possible is discussed. Finally, the motivation, objectives and structure of this study are discussed to provide the reader with an understanding of the context leading to the conduct of this work.

1.2 Flight Simulation

This section provides a short introduction to the topic of flight simulation from an historical and practical perspective. Events leading to the development of flight simulation are briefly discussed. Then, the advantages of flight simulation, which drove the development of flight simulation to its modern state, are summarized. Finally, the elements of modern day flight simulation and the need for motion cueing algorithm are introduced.

1.2.1 Flight Simulation History

In the early days of flying, teaching new pilots was an activity that was both costly and dangerous. This was in part due to the absence of training techniques other than progressive flights that would take place in actual aircraft and eventually bring a student pilot to fly on his own. Early aircraft operators acknowledged this reality and quickly foresaw a need to replace

and supplement some of the real-world flying exercises with ground-based training exercises. Ground-based flight training systems were first introduced during the First World War, in an attempt to train a larger number of pilots [1]. Most early aircraft simulation systems gave mixed results as it was unclear whether they had a positive effect on pilot training [1]. Successful aircraft simulation systems really made their appearance when computing technology allowed for basic flight equations to be solved in real time [1]. Since then, the flight simulation industry has grown significantly and there has been sustained development in ground-based flight simulators. Nowadays, a myriad of aircraft simulation systems are in use by commercial and military organizations for the training of pilots [1].

1.2.2 Advantages of Flight Simulation

Flight simulator operators benefit from a reduction in pilot training time and a massive reduction in training costs [1]. For the training of emergency procedures, flight simulators allow an unparalleled level of convenience and safety. Additionally, flight simulators are ideally suited for the training of instrument flight rules (IFR) flights and the conduct of re-currency checks [1, 2]. Furthermore, typical flight simulator environment allow for the recording of exercises and provide the users with enhanced debriefing capabilities. Flight simulators also reduce the risk of aircraft training accidents as a large portion of training can be conducted in the simulated environment.

1.2.3 Modern Flight Simulation

Typical modern full motion simulators used in pilot training, such as the ones sold by CAE Inc., use electrically or hydraulically driven six degree-of-freedom (6DOF) motion platforms and retail for several million dollars [2]. Although it has been demonstrated that in most cases a three degree-of-freedom (3DOF) motion platform is capable of producing a

simulation quality comparable to that of a 6DOF motion platform [3], there are very few commercially available 3DOF flight simulators.

Most flight simulators include several subsystems that approximate sensations felt in a real aircraft during flight. The computer generated images, the visual display, the sound system, the motion simulation system and other enhancements such as realistic controls and cockpit instrumentation all contribute to the quality and realism of the simulation. In this study, the subsystem of most interest is a 3DOF motion platform, its associated motion cueing algorithm, and the control system that lead to a more realistic sensation of flying. Modern simulators usually employ electrically or hydraulically driven motion platform to generate motion. All of these motion platforms have limited motion envelopes compared to the motion envelope of a real aircraft. The physical limits of a motion platform are referred to as the "motion envelope". Thus, a motion cueing algorithm commonly referred to as the Washout Filter (WF) algorithm is necessary to transform the aircraft's angular rates and accelerations so that the motion platform remains within its motion envelope. It has been substantiated through pilot validation testing that a properly driven motion platform for a flight simulator can greatly enhance the simulation, while an improperly cued motion platform can adversely affect the simulation [4].

1.3 RMC 3DOF Motion Platform

The Department of Mathematics and Computer Science of the Royal Military College of Canada (RMC) purchased a Triad 1500-E 3DOF motion platform from Sarnicola Simulation Systems Inc. (SSSI). This motion platform was acquired to provide students and researchers with a versatile tool capable of motion simulation in three degree-of-freedom. RMC's 3DOF motion platform, shown in Figure 1 [5], is capable of executing movements of roll, pitch and heave with a payload of up to 1500 lbs.

Figure 1 - RMC's Triad 1500-E 3DOF Motion Platform

The motion is generated by three electric motors in a triangular arrangement connected to the upper platform frame by a double shafted gear head actuator.

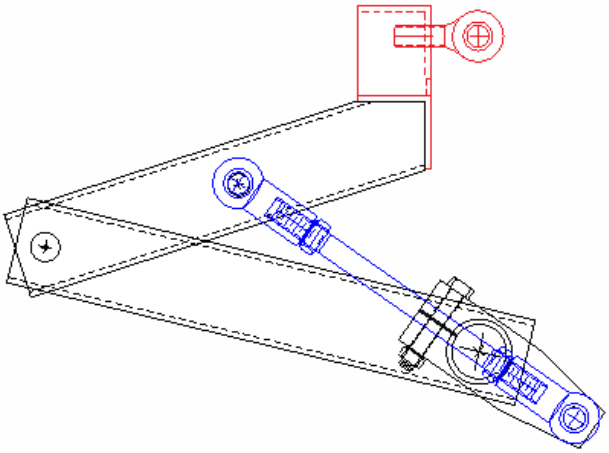


Figure 2 - Motion Platform Actuator Assembly

The actuator assembly, shown in Figure 2 and adapted from CAD drawings [6], transforms rotational motion from each of the motors into linear motion. By controlling the

length, velocity and acceleration independently for each actuator, the platform can be positioned in space and generate ranges of roll, pitch and heave motions. The motion platform displacement, velocity and acceleration limits, which determine the motion platform envelope, are given in Table 1. These recommended motion platform limits are provided by the manufacturer, SSSI [6].

Table 1- RMC's 3DOF Motion Platform Limits

	Roll	Pitch	Heave
Displacement	±25°	±30°	±0.20 m
Velocity	30°/s	30°/s	0.45 m/s
Acceleration	350°/s^2	350°/s^2	7.85 $m \bullet s^{-2}$

The displacement, velocity and acceleration of the actuators are controlled by a high performance Galil Accelera Series 30x40 motion controller. The motion controller comes with native C++ libraries and a COM interface that allows for the use of Windows programming languages such as Visual Basic (VB) and C#. It can operate in stand-alone mode or it can be interfaced with a computer through Ethernet or RS-232 serial port cables. The electric motors are driven by a Yaskawa F7 Drive acting as frequency converter with encoder feedback. In this system, the speed at which motors move is a function of the voltage frequency. The position of each actuator is extrapolated from the encoder feedback feature of the Yaskawa F7 Drive. To power the motion platform it is necessary to have a 240 volt AC, 3 phase current [6]. The Galil motion controller is powered by a regular 120 volts AC outlet.

The motion platform system comes with testing and demonstration software allowing the user to manually control the platform from a graphical user interface (GUI). This software is rudimentary and does not prevent a user from causing damage to the motion platform. This lack

of safety was experienced first hand and is discussed later in Chapter 4. Other than the demonstration package, no control software or software documentation is provided with the system. Additionally, the motion platform system was not delivered with a simulation environment or motion cueing software. However, the Galil motion controller and its libraries allow for flexibility in interfacing with the platform.

1.4 Motivation

The Canadian Forces (CF) is currently operating several simulation systems for the training of pilots and drivers of a multitude of platforms. Flight simulators are routinely used in the training and periodic skill checks of aircrew personnel. Additionally, there is an ongoing effort by the Directorate of Land Synthetic Environment (DLSE), a branch of the Land Force Doctrine and Training System (LFDTS), to assess, develop and integrate new simulation technologies in the Army []. The use of simulation in the CF training system is cost effective, safe and in most cases, is also environmentally friendly. According to the LFDTS, it is believed that an increase in CF simulation capabilities will be essential to sustain the growth of a modern and effective force.

For this reason, the motivation for this study is to provide an evaluation of the flight simulation capabilities of RMC's 3DOF motion platform as a tool to create a flight simulation environment. As there is currently no flight simulator at RMC, this research serves as a reference work in the design of such systems at RMC. The work and conclusions drawn in this study are relevant to the design and implementation of simulators using 6DOF motion platform technologies that RMC has recently acquired. Finally, as the research budgets allotted to motion simulation researches are limited, it was also of great interest to investigate whether a realistic

flight simulator can be designed and implemented using low cost and commercially available components and software.

1.5 Aim

The first aim of this research is to create a low cost flight simulator using RMC's 3DOF motion platform. The design and implementation of this flight simulator is made in order to obtain a realistic experience from RMC's 3DOF motion platform. Furthermore, the flight simulator's capabilities and degree of realism, when driven by the classical WF algorithm, are investigated. The classical WF algorithm is the most widely used algorithm for 6DOF motion platforms in the flight simulation industry [4]. It is used to filter the accelerations of a simulated aircraft so that the platform is not moved beyond its limits and so that it will "washout" or return to its central position in order to accommodate subsequent motion demands. The work completed in this research is intended to allow others to understand the underlying principles that govern RMC's 3DOF flight simulator. With that in mind, this thesis is crafted in anticipation that it will be used as a reference for other motion simulation projects to come.

This research is accomplished using a 3-step process. The first step is to review the literature and to adapt the 6DOF classical WF algorithm for RMC's 3DOF motion platform. Secondly, a flight simulator is designed and implemented to take advantage of the motion platform capabilities. Finally, an experiment is conducted in order to optimize the simulator's fidelity and to evaluate the quality of sensations it creates.

1.6 Contributions

This thesis introduces an architecture that provides a way to enable flight simulation using a low cost motion platform with commercially available components. The approach

utilized showed that an acceptable level of realism can be reached for flight simulation without the need for specifically engineered components. The flight simulator developed in this thesis shows the effectiveness of this approach in displaying the feasibility, qualities and shortfalls of such a commercially available component-based system.

More specifically, robust and flexible software designed to allow for the use of different motion platforms and virtual flight environment was implemented. The most widely used motion control algorithm for flight simulation, the classical washout algorithm, was implemented in the Simulink environment and modified for use with a 3 DOF motion platform. The use of a powerful modelling environment such as Simulink greatly simplified the traditional implementation process of the control algorithm and also simplified the fine tuning process. In this environment, a special method (polarity switch) was also developed and implemented as a mean to reduce undesirable motion platform overshooting often associated with the use of flight simulation control algorithms. This led to significant improvement of the flight simulator response.

The use of an inertial measurement unit combined with traditional techniques allowed for the effective tuning of the motion platform response. This approach also showed that there is great potential for improved algorithms using “hardware in the loop” configurations and that robust automated tuning technique could also be developed using an inertial measurement unit.

A pilot validation exercise where several pilots flew the flight simulator showed the importance of pilot evaluation by highlighting the shortfalls and positive aspects of the system. A rating questionnaire was developed to assess the performance of the flight simulator in key areas. The results of this pilot validation exercise provided valuable information that contributes the

local knowledge base on flight simulation and will allow for to the successful implementation of superior flight simulators at the Royal Military College of Canada.

Finally, the work conducted in this thesis contributed greatly in displaying the capabilities of the Royal Military College of Canada to various organisations such as Defence Research and Development Canada (DRCD), and Directorate Land Synthetic Environment (DLSE). The Canadian Association of Computer Science organized a national Computer Science Education Week between Dec. 5-11, 2010, in which 26 Canadian Universities participated; the Flight Simulation with Motion Platform out of this research was the leading event representing RMC to the nation.

1.7 Thesis Structure

The current chapter provides the reader with background information on flight simulation in general, RMC's 3DOF motion platform and the purpose of this study. The remainder of this thesis is structured as follows:

Chapter 2 reviews the classical WF algorithm in detail and explains the necessary adaptation of the algorithm from 6DOF to 3DOF.

Chapter 3 covers the considerations associated with the design of a flight simulator and explains the design of the flight simulator.

Chapter 4 describes the implementation of the flight simulator as well as the equipment and techniques used.

Chapter 5 describes the experiment conducted to optimize and study the flight simulation capabilities of the system. This chapter also contains a summary and analysis of the results.

Chapter 6 presents conclusion drawn from the research and describes area of future work to build on the results obtained herein.

The thesis is written in a logical fashion intended to lead the reader through the steps that were taken during this study.

CHAPTER 2 – MOTION DRIVE ALGORITHM AND 3DOF ADAPTATION

2.1 Introduction

A flight simulator motion platform is constrained by its physical limits and thus cannot exactly reproduce the accelerations and angular rates experienced in a real aircraft. In order to generate the suitable controls for a motion platform, it is recommended [11] to use a WF algorithm that accounts for this motion envelope. Although there are many versions of WF algorithms, the classical WF algorithm as reported by Reid and Nahon [8, 9, and 10] is used in this study because of its relative simplicity and extensive use in commercial flight simulator. Other algorithms such as the coordinated adaptative algorithm and the optimal controller algorithm exist, but are far more complex to implement and tune than the classical WF algorithm [12]. Furthermore, in a 6DOF flight simulator study by the same authors [4], the classical WF algorithm has been reported to achieve very good results when compared to the other algorithms.

In this chapter, the reference frames and axis system employed in the formulation of the motion drive algorithm is introduced. This chapter also consist of a review of the 6DOF classical WF algorithm as developed and studied extensively at the University of Toronto Institute for Aerospace Studies (UTIAS) [4, 8, 9, 10, 11 and 12]. As such, it will be reviewed in detail for each channel. Finally, the modifications required to implement the classical WF algorithm to

RMC's 3DOF motion platform are discussed in this chapter as they are better understood in the context of the original algorithm.

2.2 Reference Frames and Axis System

When discussing any WF algorithm it is important to define the reference frames and axis system that are used in the formulation of the algorithm to describe the relative position of objects in space. In this case, the objects of interest are the aircraft and the flight simulator. As such, the reference frames and axis system as reported by [8, 9, and 10] are used in this study and can be seen in Figure 3.

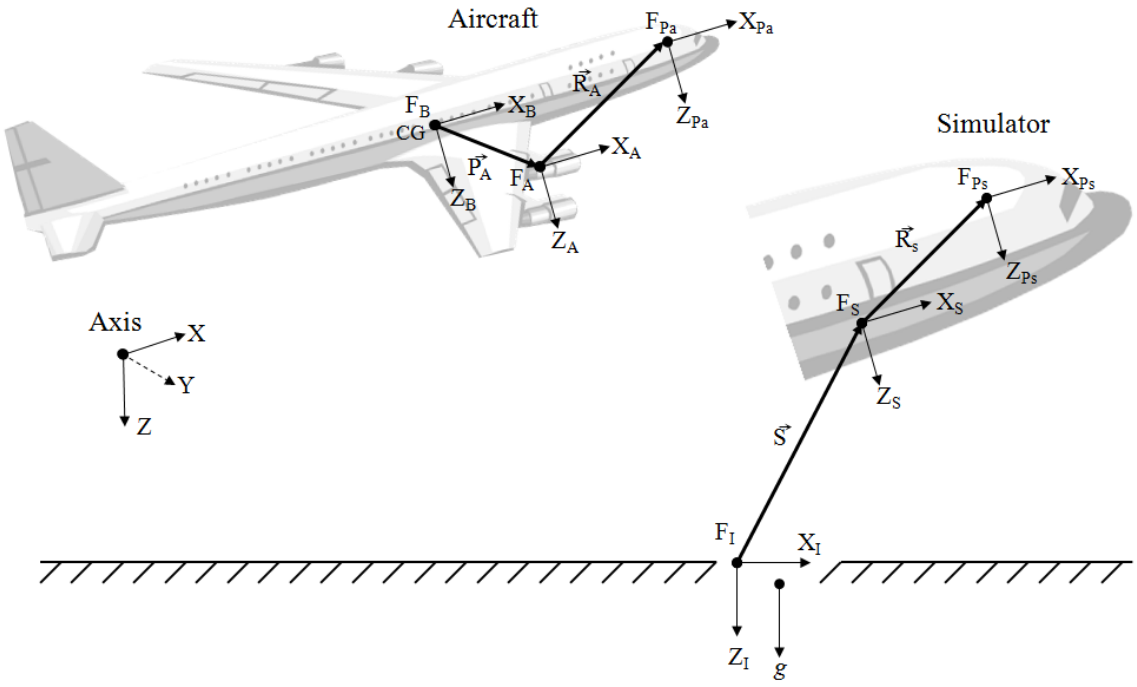


Figure 3 - Reference Frames and Axis System

Frame F_I, the inertial reference frame, is Earth-fixed with its z axis parallel to the earth gravity, g. F_I is also associated with both the aircraft and the simulator. The right hand orthogonal axis system is used in this study. In this axis system, the x axis can be described as pointing towards the nose of the aircraft and the y axis, orthogonal to the x and z axes, as

pointing out the right wing of the aircraft. The z axis is described as pointing downward with respect to the aircraft. The direction of a positive rotation around each axis of the coordinate system is determined by using the right hand rule. A simulator reference frame, F_S, is geometrically centered on the upper frame of the motion platform. The location of F_S with regards to F_I is represented by the vector $\vec{S}$, which connects the origin of frame F_I to the origin of frame F_S. Frame F_{Ps}, the simulator pilot reference frame, is located at the pilot's head between his left and right vestibular systems. Furthermore, F_{Ps} is parallel to F_S and both frame's origins are connected by $\overrightarrow{R_S}$. Vector $\overrightarrow{R_S}$ represents the position of the pilot's head with regards to the simulator reference frame F_S. F_{Pa}, the aircraft pilot reference frame, is located and oriented exactly like F_{Ps} with respect to the cockpit. In addition, aircraft reference frame F_A shares the same orientation and relative cockpit location as the simulator reference frame F_S. Frame F_A is required in order to compare the simulated aircraft motions to that of the motion platform in frame F_S. In other words, $\overrightarrow{R_A}$ and $\overrightarrow{R_S}$ are equal in magnitude and parrallel. Finally, aircraft reference frame F_B is fixed at the aircraft's center of gravity with its x axis pointing forward (towards the nose) and its z axis point downward with respect to the aircraft. Vector $\overrightarrow{P_A}$ connects the origins of frames F_B and F_A in space.

2.3 Classical Washout Filter Algorithm

The classical WF algorithm, shown in Figure 4, is comprised of a translational motion channel, a tilt-coordination channel and a rotational motion channel. Due to the limited motion envelope of any motion platform, it is not possible to reproduce constant (low frequency) translational or rotational motions as they would constantly drive the motion platform to its physical limits. As such, the classical WF algorithm filters out lower frequency motions in both

its translational and rotational motion channels. An approximation of low frequency translational motions is performed in the tilt-coordination channel where the motion platform is oriented slightly off the gravity vector in order to create a translational motion sensation.

The inputs to the classical WF algorithm are the aircraft accelerations, $\overrightarrow{f_{AA}}$ and the aircraft angular rates, $\overrightarrow{\omega_{AA}}$ where $\overrightarrow{f_{AA}} = (a_x, a_y, a_z - g)$ and $\overrightarrow{\omega_{AA}} = (p, q, r)$. In this formulation a_x, a_y *and* a_z are the accelerations along the *x*, *y* and *z* axes respectively and *g* is the acceleration due to gravity ($9.8\ m{\bullet}s^{-2}$). The variables *p*, *q* and *r* represent the Euler angle rates at which the aircraft is moving around each of its axis from *x* to *z* in order.

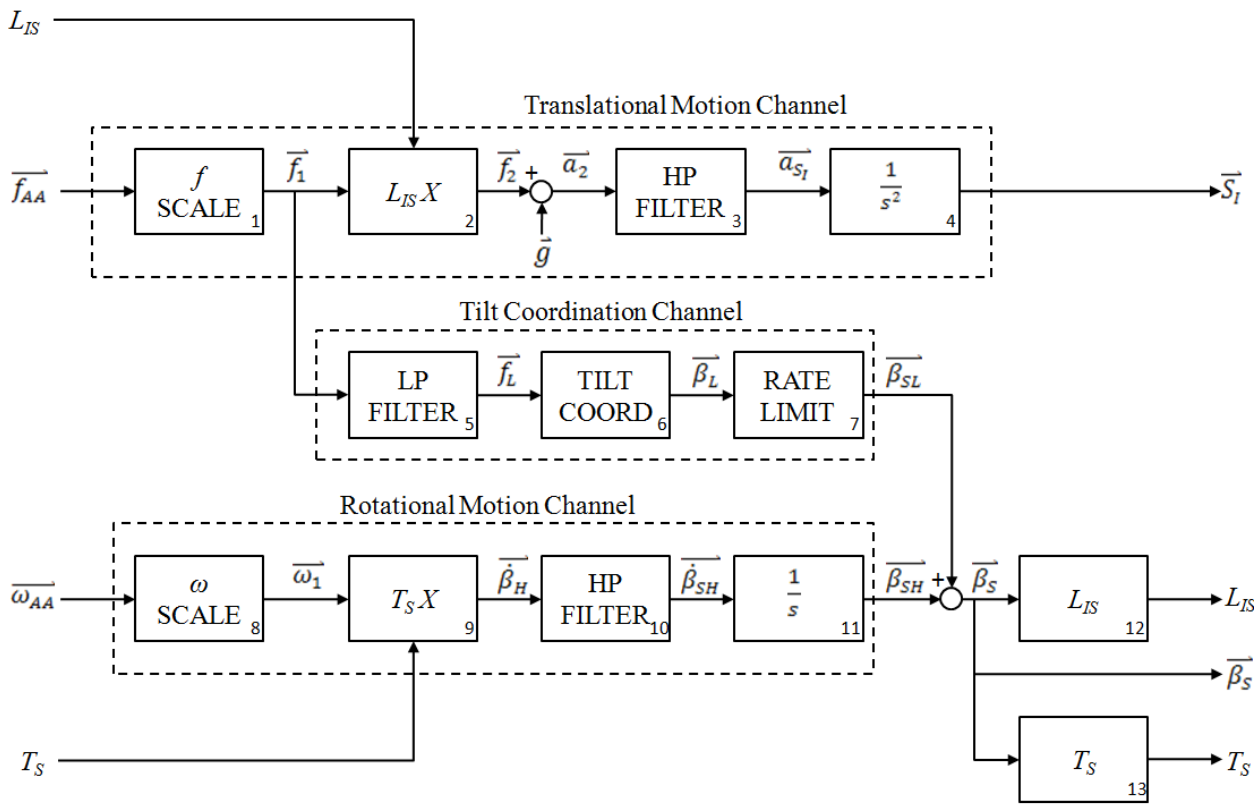


Figure 4 - Classical WF Algorithm [6]

In this study, aircraft accelerations are expressed using the metric system in $m{\bullet}s^{-2}$ and the angular rates are expressed in $deg{\bullet}s^{-1}$. The outputs of the classical WF algorithm are the

simulator displacement, $\overrightarrow{S_I}$ where $\overrightarrow{S_I} = (d_x,\ d_y, d_z)$ and the simulator Euler angles, $\overrightarrow{\beta_S}$ where $\overrightarrow{\beta_S} = (\phi, \theta, \psi)$. In this formulation, *dx*, *dy* and *dz* are the displacement of the motion platform for each axis. The variables $\phi, \theta\ and\ \psi$ represent the motion platform Euler angles with regards to the earth reference frame F_I. When combining $\overrightarrow{S_I}$ and $\overrightarrow{\beta_S}$, it is possible to obtain the final position of the motion platform with respect to F_I. From this information, we can derive the appropriate actuator lengths, speeds and accelerations required to achieve this position. In the sections to follow, each channel and its respective signals of the algorithm are discussed in details. Finally, the term "Box" followed by a number used in the following sections refers to the corresponding function block of the classical WF algorithm as illustrated in Figure 4.

2.3.1 Translational Motion Channel

The translational motion channel, consisting of Boxes one to four of Figure 4, transforms the aircraft accelerations $\overrightarrow{f_{AA}}$ into simulator translational displacement $\overrightarrow{S_I}$. In Box one, accelerations are first scaled down to be within the physical limits of the motion platform being used and limited in function of the motion platform physical limits (additional information can be found in Section 2.3.4). The scaled down and limited accelerations are represented by the intermediate variable $\overrightarrow{f_1}$ that is used later in the tilt-coordination channel. In Box two, the accelerations are transformed from reference frame F_A to reference frame F_I using the Euler rotation matrix L_{IS} (given by Equation 2.1 [8]). The intermediate earth referenced accelerations derived from Box two are represented by $\overrightarrow{f_2}$.

$$L_{IS} = \begin{bmatrix} cos\theta\ cos\psi & sin\phi\ sin\theta\ cos\psi - cos\phi\ sin\psi & cos\phi\ sin\theta\ cos\psi + sin\phi\ sin\psi \\ cos\theta\ sin\psi & sin\phi\ sin\theta\ sin\psi + cos\phi\ cos\psi & cos\phi\ sin\theta\ sin\psi - sin\phi\ cos\psi \\ -sin\theta & sin\phi\ cos\theta & cos\phi\ cos\theta \end{bmatrix} \quad (2.1)$$

The local gravity vector $\vec{g}$ is then added to the accelerations producing intermediate specific accelerations represented by $\overrightarrow{a_2}$. In this formulation, $\overrightarrow{f_2}$ represents the scaled down and limited aircraft specific accelerations in the earth reference frame F_I. Next, $\overrightarrow{a_2}$ is obtained with the addition of the local gravity vector to $\overrightarrow{f_2}$. As can be seen in Box twelve, the Euler rotation matrix L_{IS} is formed using the current simulator orientation given by Euler angles represented by $\overrightarrow{\beta_S}$. In Box three and four, the accelerations are high-pass filtered and integrated twice to obtain, $\overrightarrow{S_I}$, the simulator's translational displacements. High pass filtering is necessary to eliminate the low frequency accelerations that would displace the actuators to their physical limits. The intermediate variable $\overrightarrow{a_{S_I}}$ represents the high-pass filtered accelerations.

2.3.2 Rotational Motion Channel

The rotational motion channel, consisting of Boxes eight to eleven, transforms the aircraft angular rates $\overrightarrow{\omega_{AA}}$ into simulator high-frequency Euler angles $\overrightarrow{\beta_{SH}} = (\phi\beta_H,\ \theta\beta_H,\ \psi\beta_H)$. In Box eight, the aircraft angular rates are scaled down and limited in function of the motion platform physical limits (additional information can be found in Section 2.3.4). The scaled down and limited angular rates are represented by the intermediate variable $\overrightarrow{\omega_1}$. In Box nine, the angular rates are transformed into Euler angle rates, $\overrightarrow{\dot{\beta}_H}$, using the transformation matrix T_S (given by

Equation 2.2 [8]) As can be seen in Box thirteen, the transformation matrix T_S is formed using the current simulator orientation given by Euler angles $\overrightarrow{\beta_S}$.

$$T_S = \begin{bmatrix} 1 & sin\phi\ tan\theta & cos\phi\ tan\theta \\ 0 & cos\phi & -sin\phi \\ 0 & sin\phi\ sec\theta & cos\phi\ sec\theta \end{bmatrix} \tag{2.2}$$

In Boxes ten and eleven the Euler angle rates $\overrightarrow{\dot{\beta}_H}$, are high-pass filtered and integrated once to obtain the simulator high frequency Euler angles $\overrightarrow{\beta_{SH}}$. Filtering out low frequency rates is required as they would drive the motion platform to its limit. The intermediate variable $\overrightarrow{\dot{\beta}_{SH}}$ represents the high-pass filtered Euler angle rates before they are integrated in Block 11.

2.3.3 Tilt-Coordination Channel

The tilt-coordination channel transforms the scaled aircraft accelerations into $\overrightarrow{\beta_{SL}}$, the simulator low frequency Euler angles. The low frequency Euler angles $\overrightarrow{\beta_{SL}}$ represent the angles at which the simulator cab must be positioned to effectively simulate continuous translational accelerations. They are obtained in the tilt coordination channel from the low frequency (constant) accelerations of the aircraft. Due to the limited translational motion envelope of motion platforms it is hard to simulate continuous translational motion in the *x* and *y* axis as sensed by an aircraft pilot. However, by slightly tilting the simulator cab with appropriate visual cues, it is possible to orient to the local gravity vector in such a way that the simulator generates a translational motion feeling for the pilot. This process, called tilt-coordination is commonly used in flight simulators [8]. When using tilt-coordination to simulate translational accelerations, it is important to limit the angular velocity at which the simulator cab tilts. Indeed, as reported by [3], vestibular motion detection threshold values range between 2.6 to 3.6 $deg \bullet s^{-1}$. Limiting the simulator cabin tilt-coordination rates below these threshold values is paramount to achieve

realistic simulation. In the example shown in Figure 5, the simulator cab is slowly tilted about its *y* axis until it reaches approximately six degrees. Then, the pilot experiences an approximate translational acceleration $9.81\, m \cdot s^{-2} * \sin(6) \approx 1\, m \cdot s^{-2}$. The a_z component of the motion simulator acceleration is affected during this process. However, if the tilt-coordination angles are sufficiently small, this reduction has a negligible effect on the perceived overall motion [8].

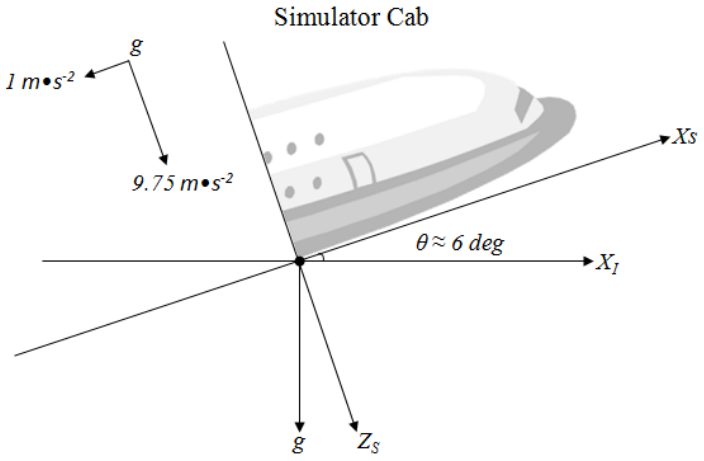


Figure 5 - Example of Tilt-Coordination

In Box five of the tilt-coordination channel, the low frequency aircraft accelerations are extracted using a low-pass filter and held in the intermediate variable $\overrightarrow{f_L}$. Tilt-coordination takes places in Box six where $\overrightarrow{\beta_L} = (\phi\beta_L,\ \theta\beta_L,\ \psi\beta_L)$ is a vector containing the low frequency tilt-coordination Euler angles. The low frequency tilt-coordination Euler angles of low performance aircrafts with low speed and manoeuvrability such as airliners are described in Equations 2.3 to 2.5 [8].

$$\phi\beta_L = -\frac{f_{Ly}}{g} \tag{2.3}$$

$$\theta\beta_L = \frac{f_{Lx}}{g} \tag{2.4}$$

$$\psi\beta_L = 0 \tag{2.5}$$

In Box seven, the low frequency tilt-coordination Euler angles are rate limited to be below the vestibular motion detection threshold values. As described by Equation 2.6, the output of the tilt-coordination channel is combined with the output of the rotational channel to form the simulator Euler angles where $\overrightarrow{\beta_{SL}} = (\phi,\ \theta,\ \psi)$.

$$\overrightarrow{\beta_{SL}} + \overrightarrow{\beta_{SH}} = \overrightarrow{\beta_S} \tag{2.6}$$

2.3.4 Scaling and Limiting

The scaling and limiting of aircraft accelerations and angular rates is essential to generate motion platform commands that are proportional to those of the simulated aircraft while at the same time suited for the physical limitations of the motion platform.

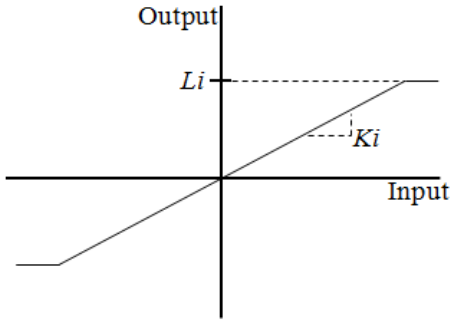


Figure 6 - Scaling and Limiting [6]

The process used in the classical WF algorithm is illustrated by Figure 6 where L_i is the limiting value and the slope K_i is the scaling value [6]. The scaling process can also be interpreted as a gain function as described in Equation 2.7. Furthermore, the limiting process can be interpreted as a conditional statement as described in Equation 2.8. In the case of scaling for the aircraft accelerations $\overrightarrow{f_{AA}}$, the input variable in Equations 2.7 and 2.8 is represented by the specific accelerations (the resulting accelerations when the effect of gravity is removed (a_z only)). This is required because the simulator cab is under the constant effect of the local $\vec{g}$, thus only

the aircraft accelerations that depart from $\vec{g}$ are simulated [8]. Scaling $\overrightarrow{f_{AA}} - \vec{g}$ would lead to a false motion cue when the local gravity vector is added after Block 2 of the classical WF algorithm. The aircraft angular rates $\overrightarrow{\omega_{AA}}$ are taken as is and scaled according to Equations 2.7 and 2.8.

$$Input * K_i = Scaled\ Input \tag{2.7}$$

$$Limited\ Input = \begin{cases} Input & If\ |Input| < L_i \\ L_i\left(\frac{Input}{|Input|}\right) & Else \end{cases} \tag{2.8}$$

2.3.5 Filtering

Filtering plays an essential role in the classical WF algorithm. Indeed, due to the physical limitations of motion platforms it is necessary to attenuate the low frequency accelerations that tend to drive motion platforms to their limits [12]. High frequency accelerations are usually of short duration and do not drive the simulator to its limits as easily. Another important role of the filters is to slowly and continuously bring the motion platform back to its central location [12]. The classical WF algorithm uses both low pass and high pass filters to accomplish this task. Equations 2.9 will be recognized as that of a 2nd order high pass filter, 2.10 as that of a 1st order high pass filter and 2.11 as that of a 2nd order low pass filter [11]. These equations describe the Laplace domain transfer functions of the different filters used in each channels of the classical WF algorithm.

Translational channel filter (Box 3): $$\frac{s^2}{s^2+2\zeta\omega_n s+\omega_n^2} \tag{2.9}$$

Rotational channel filter (Box 10): $$\frac{s}{s+\omega_n} \tag{2.10}$$

Tilt-coordination channel filter (Box 5): $$\frac{\omega_n^2}{s^2+2\zeta\omega_n s+\omega_n^2} \quad (2.11)$$

Choosing the appropriate parameters for the filters can only be done with a trial and error tuning process combined with pilot feedback [12]. Several false cues can be generated by improperly tuned filters [12]. As such, the motion platform designer must perform several tests by isolating each filter and ensuring a proper behaviour depending on the application. The absence of a filter damping ratio ζ (a parameter used to characterize the frequency response of filters) for first order filters makes them easier to fine tune. Second order filters provide a much stronger attenuation power, but can lead to other problems such as overshooting (when the signal exceeds its final target value during a transition from one value to another). Table 2 shows the filter parameters used with the classical WF algorithm that received the most favourable response from pilots during a study at the UTIAS [11].

Table 2 - Filter Parameters with Most Favourable Pilot Response (UTIAS) [11]

Filter	Order	ω_n	ζ
x translational high-pass	2	2.5	1.0
y and z translational high-pass	2	4.0	1.0
All rotational high-pass	1	1.0	-
x translational low-pass	2	5.0	1.0
y translational low-pass	2	8.0	1.0

These values are a good starting point for a study, but would need to be fine tuned after evaluating their behaviour and their effects on each channel.

2.4 Three Degree-of-Freedom Adaptation

In order to use the classical WF algorithm with RMC's 3DOF motion platform, some modifications are required. First, RMC's 3DOF motion platform is not capable of generating surging and swaying motions characterized by translational displacement in the *x* and *y* axis respectively. The platform can only generate heave motions characterized by translational displacements in the *z* axis. Furthermore, the platform is limited in its rotational displacement capabilities as it is not capable of generating yaw motions characterized by a rotation about its *z* axis. The platform is only capable of generating roll and pitch motions characterized by rotations about its *x* and *y* axis respectively.

Despite those limitations, the tilt-coordination channel of the classical WF algorithm allows for an approximation of the desired translational effects in the *x* and *y* axis by tilting the simulator's cabin towards these axes as described in Section 2.2.3. In addition, the absence of a yaw motion in the simulation of fixed wing aircrafts is not a significant problem as there is usually little yaw motion in normal flight. As such, for the simulation of low performance aircraft, the performance of 3DOF motion platforms is comparable to that of 6DOF motion platforms [3].

2.4.1 Translational Motion Channel

As suggested by [11], the optimal scaling factor for most commercial 6DOF simulators is of 0.5. However, depending on the behaviour of the translational channel high pass filter utilized as well as RMC's 3DOF specific motion envelope, this coefficient may need to be modified to suit this study. An appropriate scaling factor will be chosen during the experimental phase of this study, in which fine tuning of the algorithm will occur. According to the platform manufacturer,

SSSI [6], the *z* axis accelerations of RMC's 3DOF are limited to a maximum of 7.85 m•s^{-2} due to the physical limitations of the platform.

RMC's 3DOF motion platform is only capable of translational displacements in the *z* axis. For this reason, the *x* and *y* translational accelerations are not required in the calculation of the simulator translational displacement $\overrightarrow{S_I}$. However, the *x* and *y* translational accelerations are still required for the tilt-coordination channel.

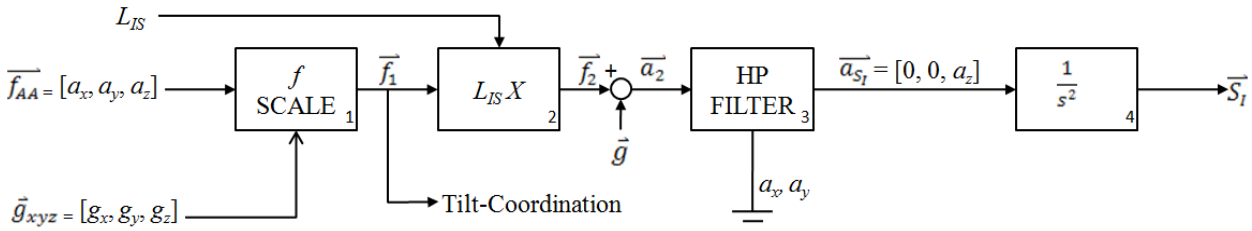


Figure 7 - 3DOF Translational Motion Channel

As can be seen in Figure 7, the unnecessary accelerations are discarded in the high-pass filtering step (Box three), but all the accelerations make their way to the tilt-coordination channel. Since the *x* and *y* translational acceleration components are not required, only the *z* component is high-pass filtered using a second order filter. In addition, it can be seen that the vector $\vec{g}_{xyz}$ is added to Block 1. This step is necessary for this study since the aircraft accelerations that are obtained from Microsoft Flight Simulator X (FSX), the virtual environment used in this study, do not include the force of earth gravity. The algorithm is designed to receive aircraft accelerations that include the effect of gravity. Additional information on this topic and the solution to this problem is available in Section 4.2.4. Filtering of the low frequency accelerations only occurs after the accelerations are brought to the earth fixed reference frame F_I, using the Euler transformation matrix L_{IS}. High pass filtering can be done in both the F_S or F_I reference frames, but each location has its advantages and disadvantages. When filtering in either F_S or F_I, false

offsets or incorrect motion cues will be generated. However, according to the study conducted by [8], filtering in F_I is preferable as the adverse effects associated with this frame are less than that of filtering in F_S.

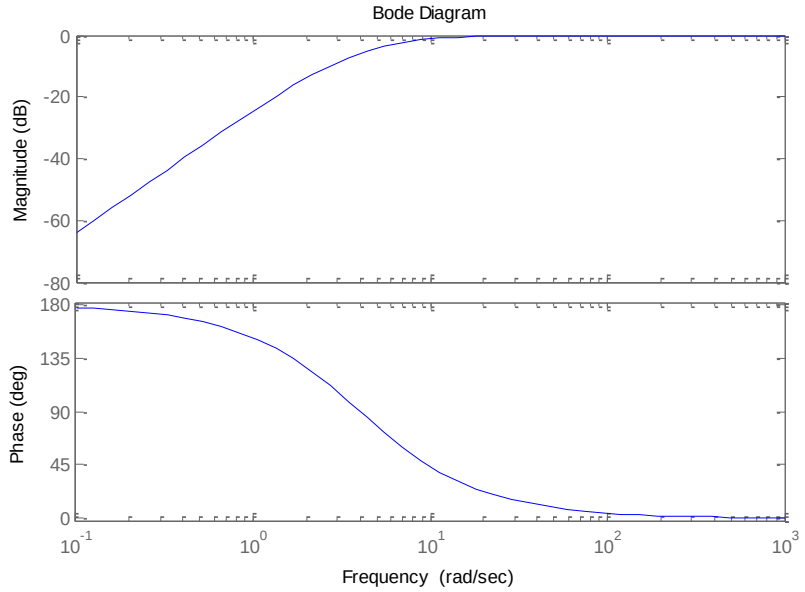


Figure 8 - Translational *z* Component High-Pass Filter (ω_n= 4 *rad/sec*)

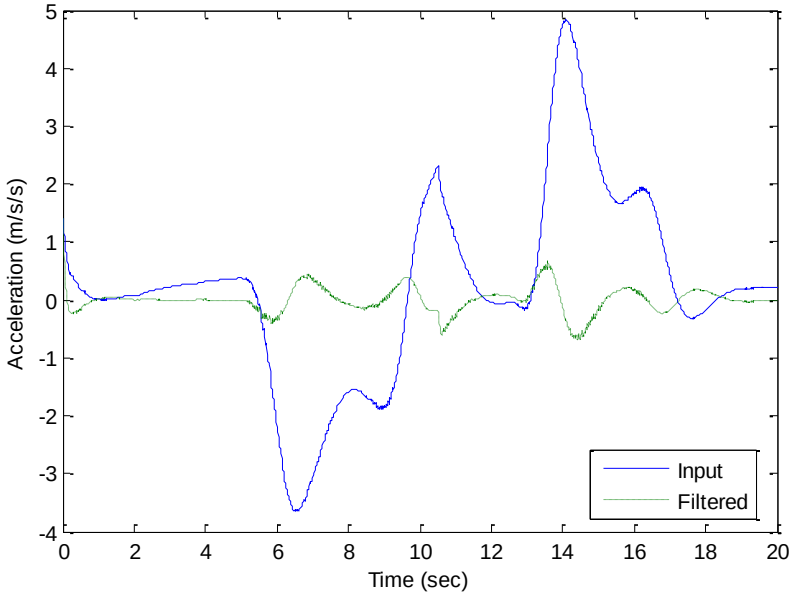


Figure 9 – Translational Channel High-Pass Filter Sample Run (ω_n= 4 rad/sec)

Figure 8 shows the Bode plot of the high-pass filter used to filter the *z* translational acceleration component where the frequency is in $rad \bullet s^{-1}$ (the standard unit for bode plots). As can be seen from the gain magnitude plot, the magnitude of a signal is progressively reduced by a negative gain as its frequency lowers. This has a scaling effect on the signal. For this reason, fine tuning in conjunction with the scaling values of the translational motion channel is indicated to obtain the overall desired scaling effect. The Phase plot is commonly used to illustrate the frequency response phase shift. The general behaviour of the *z* component high-pass filter can be seen in Figure 9 where the results of a sample run are shown. As can be seen from this sample run, the low frequency accelerations are attenuated and the general amplitude of the signal is reduced.

2.4.2 Rotational Motion Channel

In the rotational motion channel, the angular rates are scaled down by a factor which is determined in the fine tuning phase. As discussed in the literature [11] a scaling value of 0.5 usually represents a good starting point. The angular rates are limited to 30 degree per second due to the physical limitations for safe operation of the motion platform as reported by the manufacturer (SSSI [6]).

RMC's 3DOF motion platform is only capable of rotational motion about the *x* and *y* axis. As such, rotational motion about the *z* axis is not necessary in the calculation of the high frequency Euler angles $\overrightarrow{\beta_{SH}}$ for the 3DOF version of the algorithm. The *r* component of Euler angle rates must be zeroed to prevent the propagation of *z* axis rotational motion further into the algorithm where the Euler transformation matrix L_{IS} is generated. This step is necessary for the 3DOF algorithm because matrices L_{IS} and T_S are formed from the output of the rotational motion channel and are both intended to reflect the simulator's current position. Thus, the propagation of

a *z* axis rotational motion component could cause the matrices to report unfeasible simulator positions including yaw, which would be detrimental to the simulation. As can be seen in Figure 10, the unnecessary *r* component (representing *z* axis rotational motion) of Euler angle rates $\dot{\vec{\beta}}_H$ is discarded in the high-pass filter step (Box ten).

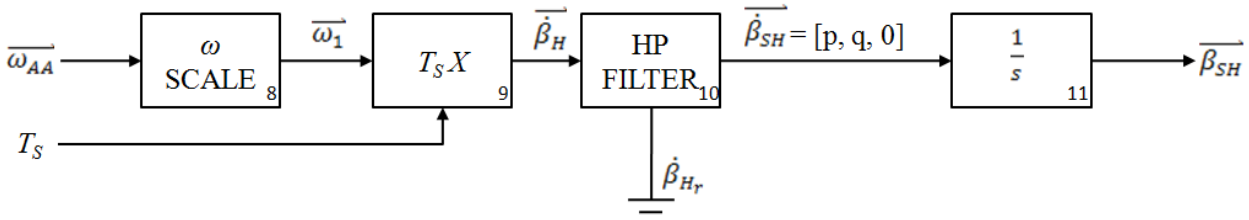


Figure 10 - 3DOF Rotational Motion Channel

In the rotational motion channel, Euler angle rates are filtered using a first order high-pass filter. As discussed in Section 2.4.1, the high pass filtering in the rotational motion channel is also performed in frame F_I in order to minimize offsets and errors. Figure 11 shows the Bode plot for the first order high-pass filter used to filter Euler angle rates.

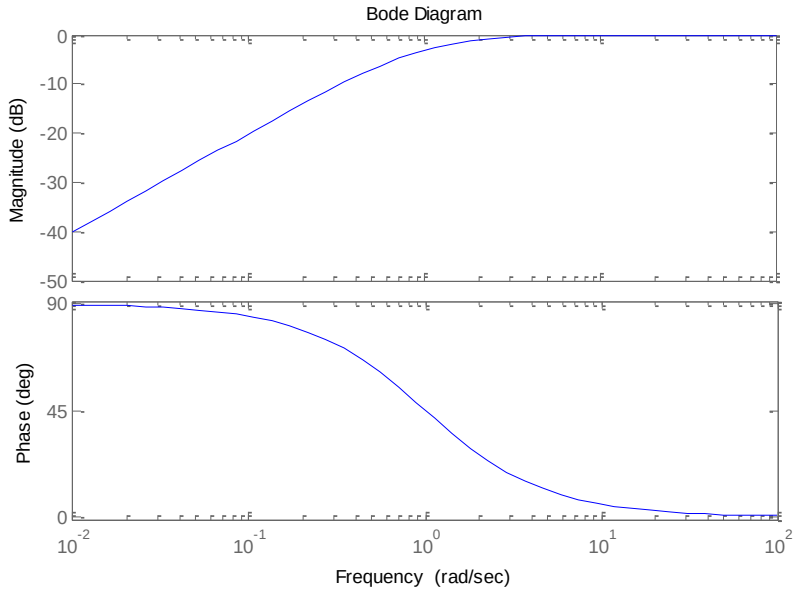


Figure 11- Rotational Channel High-Pass Filter (ω_n = 1 rad/sec)

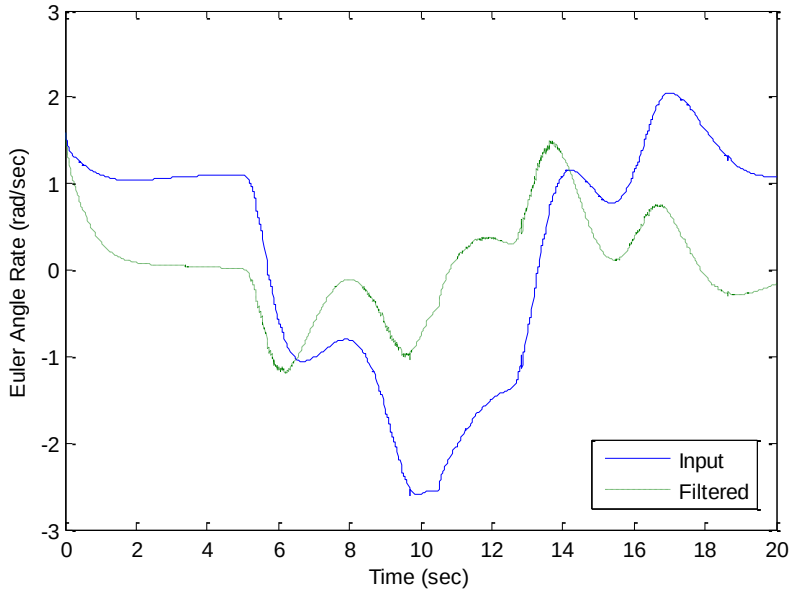


Figure 12 - Rotational Channel High-Pass Filter Sample Run (ω_n = 1 rad/sec)

As can be seen from the gain magnitude plot, a negative gain is applied to the signal as its frequency lowers. For this reason, fine tuning with the scaling values of the rotational channel is indicated to obtain the overall desired scaling effect. The general behaviour of the rotational motion channel high-pass filter can be seen in Figure 12 where the results of a sample run are shown. As can be seen from this sample run, the amplitude of the Euler angle rate signal is attenuated. It can also be observed that the attenuation is less than that of the high pass filter in the translational motion channel. This can be attributed to the slope of the gain magnitude plot, which is less pronounced for this filter when compared to the filter used in the translational motion channel.

2.4.3 Tilt-Coordination Motion Channel

The tilt-coordination channel relies on rotation about the *x* and *y* axis, which RMC's 3DOF can generate. As such, the tilt-coordination channel does not require any modification to work with RMC's 3DOF motion platform. In this study, the tilt-coordination coefficient is

determined experimentally and kept below the suggested threshold values [11]. This process is covered in Chapter 5. The *x* component low-pass filter Bode plot can be seen in Figure 13.

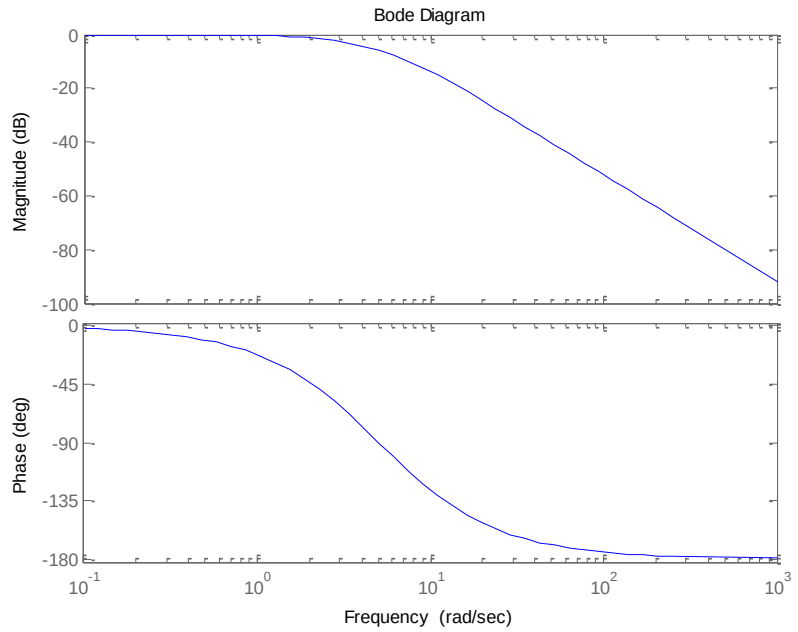


Figure 13 – Tilt-Coordination Channel *x* Component Low-Pass Filter (ω_n = 5 rad/sec)

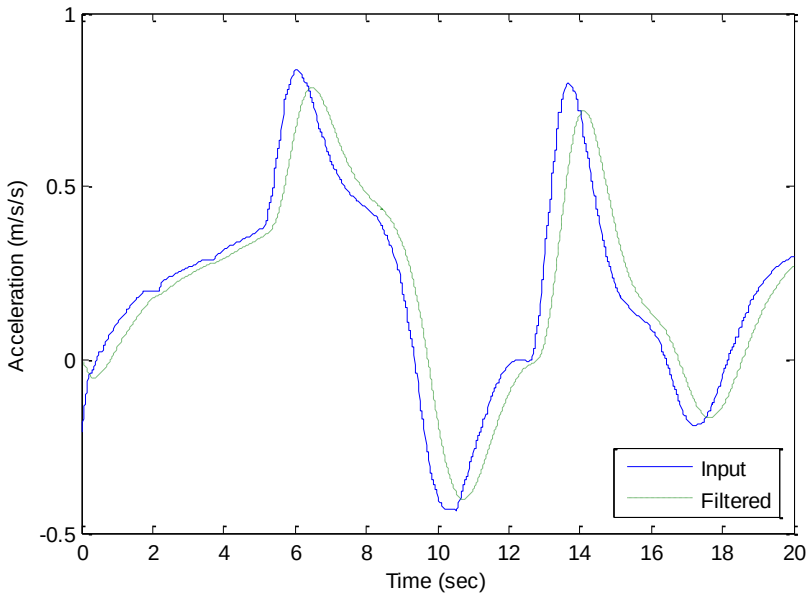


Figure 14 - Tilt-Coordination Low-Pass Filter *x* Component Sample Run

As can be seen in the gain magnitude plot, as the signal's frequency increases a negative gain is applied to reduce its amplitude. As can be seen from Figure 14, where an aggressive run with low frequency components is conducted, the filtered accelerations remain very close to the unfiltered accelerations. The parameters of the filter can be adjusted to increase the attenuation of the signal as required. During the tuning phase described in Chapter 5, filters are tuned in conjunction with the rate limiter of the tilt-coordination channel to achieve the desired effect. Although a slightly higher filter break frequency (ω_n = 8 rad/sec) is used for the *y* component low-pass filter, it exhibits a behaviour very close to that of the *x* component filter; the signal simply begins to be attenuated at a slightly higher frequency. For this reason, only the *x* component low-pass filter was discussed in this section.

2.5 Summary

This chapter provided insight into the reference frames and axis system used throughout this study. The classical WF algorithm as reported by Reid and Nahon [8, 9, 10] has been reviewed with an emphasis on each individual channels. The limitations of RMC's 3DOF motion platform were discussed along with the necessary modifications for the implementation of the classical WF algorithm. Additionally, the scaling, limiting, filtering and actuator motion calculation steps were discussed for the 3DOF version of the algorithm. With an understanding of the algorithm and limitations of RMC's 3DOF motion platform, a flight simulator can be designed and implemented.

CHAPTER 3 – DESIGN

3.1 Introduction

With an understanding of flight simulation, RMC's 3DOF motion platform and the classical WF algorithm, a plan can be developed to build a flight simulator that takes advantage of the available hardware. As this study is aimed at determining the capabilities of RMC's 3DOF motion platform and its associated classical WF algorithm, it is necessary to design a flight simulator that facilitates the conduct of the study while taking advantage of the motion platform. As such, the design of RMC's 3DOF flight simulator is done through a systematic approach keeping in mind a series of pre-defined design objectives that are conducive to the accomplishment of this goal. Furthermore, as resources are limited and the equipment has specific limitations, the design process is also influenced by a set of constraints that are inherent to the system. In the design process, available equipment, design objectives and constraints are taken into consideration in order to implement a flight simulator that effectively balances all criterions while meeting the study goals.

In this chapter, the objectives associated with the design of RMC's 3DOF flight simulator are discussed. Similarly, the constraints which must be considered during the design are discussed. In light of the design objectives and constraints, the design choices and components selected for the flight simulator are discussed in detail.

3.2 Design Objectives

An important step in design is to clarify the design objectives. When creating a flight simulator, one of the most basic design objectives is to create conditions where a flight environment is approximated to a degree that is deemed satisfactory for the conduct of specific activities [8]. In this case, the activities that are of interest are the regular flight operations of aircraft that exhibit low translational accelerations and angular rates. Indeed, due to the platform motion envelope and limitations, it would be unreasonable to attempt the simulation of very agile and aggressive aircraft such as fighter jets. Other design factors that may pertain to the available equipment for simulator subsystems, such as safety, maintenance and user interactions, must also be taken into consideration when defining design objectives. By defining design objectives, it is necessary to begin a research phase in which the literature and the industry are investigated in order to meet the design objectives. As dictated by mainstream design processes [13], a thorough investigation of proposed design objectives is necessary to ensure feasibility of design and to allow for the selection of appropriate components. An objective tree [13] is a useful and clear method for visualizing an objective statement. It is a way to display all objectives and sub-objectives of a given project into their hierarchical levels. A diagram of the top level objective tree used in the design of RMC's 3DOF motion simulator can be seen in Figure 15. Note that each of the six objectives in this tree branch into sub-objectives that will be discussed in the sections to follow.

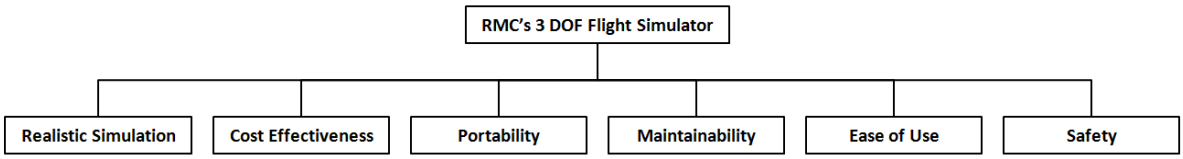


Figure 15 - Objective Tree for RMC's 3DOF Flight Simulator

The specific design objectives of RMC's 3DOF flight simulator were derived from discussions with thesis surpervisors about the desirable characteristics of a flight simulator for this study. Driven by those characteristics, the design of RMC's 3DOF flight simulator was made while keeping the six main design objectives of Figure 15 in mind. One of the primary design objectives is to create a flight simulator that is capable of approximating a real flight environment with the highest possible quality given the limited resources available. Additionally, the simulator must meet other design objectives that pertained to cost effectiveness, software portability, maintainability, ease of use and safety. In the sections to follow, each of the design objectives are discussed.

3.2.1 Achieving a Realistic Simulation

One of the primary objectives driving the design of RMC's 3DOF flight simulator is the achievement of a realistic simulation. It is of importance to generate motion, visual and audio signals that are conducive to a high degree of realism. Indeed, the generation of visual stimuli is considered to have the greatest effect on the overall realism of the simulation along with motion stimuli [14]. In designing a flight simulator, it is possible to spend thousands of dollars on systems that will generate the aforementioned stimuli to various degrees of quality. However, regardless of the quality of the systems used in the flight simulator, the most fundamental element remains the virtual environment. Great care must be taken in selecting flight simulator software that creates a virtual environment offering high quality audio-visuals generation as well as a realistic physics of flight. Once a high quality virtual environment has been selected, it is important to ensure that high quality motion simulation and audio-visual systems bear fidelity to the virtual environment and effectively immerse the pilot inside the simulation. Of particular importance to the motion simulation aspect of flight simulation is the proper motion cueing.

Indeed, it has been demonstrated that poorly cued motion is detrimental to flight simulation [4]. The classical WF algorithm, the most widely used algorithm in the flight simulation industry, has been reported to exhibit excellent motion cueing capabilities [11]. However, the filters used in the algorithm must be properly fine tuned, in accordance with the motion hardware, in order to provide the best possible motion cues [12]. Figure 16 shows the specific design sub-objectives relating to the design objective of achieving a realistic simulation.

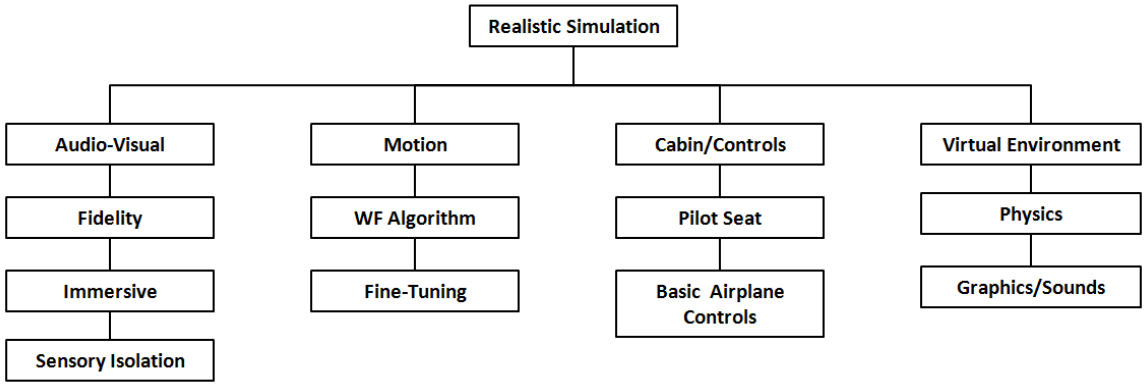


Figure 16 - Realistic Simulation Design Sub-Objectives

Various systems such as screens, projectors and head mounted displays are commercially available to generate visual cues. Factors such as image quality, screen size and resolution have an impact on the cost of such devices. It is also important to eliminate any fixed reference points when trying to immerse a passenger in a virtual environment. Without proper visual isolation (isolating the passenger's vision from the motionless surroundings), the pilot could receive misleading cues from his/her ambient (peripheral) vision from the surroundings of the simulator.

It is possible to deliver audio stimuli to the passenger by either an ambient or headset system. To provide a realistic simulation, the occupant of the flight simulator should be isolated from outside noises. Both ambient of headset systems are suitable depending on the presence of a cabin or not.

Finally, the simulation can be further enhanced by providing a realistic cabin environment and controls to the passenger. Various systems are available and typically range from simple consumer level gaming joysticks, throttles and rudders to fully replicated and functional aircraft cabins with simulated instrumentation panels. Some of the most realistic controls can provide tactile sensation in the form of force-feedback from the simulation environment depending on the situation.

For this study, it is felt that at a minimum, the controls necessary to operate an aircraft should be present along with a seat for the pilot to be properly seated. A seat providing adjustments is desirable as it will allow individuals of different heights to be seated comfortably.

3.2.2 Cost Effectiveness

As resources allocated to this study are not unlimited, cost effectiveness is an important objective driving the development of RMC's 3DOF flight simulator. Furthermore, as highlighted in Section 1.4, an objective of this research is to design a flight simulator at a relatively low cost while maintaining a high level of realism. The conduct of a cost-benefit analysis is an important step in achieving cost effectiveness. Indeed, the process of cost-benefit analysis consists of comparing the amount of money spent to the benefits that will be achieved through the spending of this money to acquire a specific equipment or software. This can only be achieved by carefully identifying the sub-systems of a flight simulator that bring the greatest contribution to realism (that will provide the greatest benefits). The specific design sub-objectives relating to cost effectiveness can be seen in Figure 17. It is important to thoroughly investigate the commercially available subsystem components to gain a market perspective on their quality, cost and availability. Finally, an aspect of equipment that brings added benefits to a design is the polyvalence of the equipment selected.

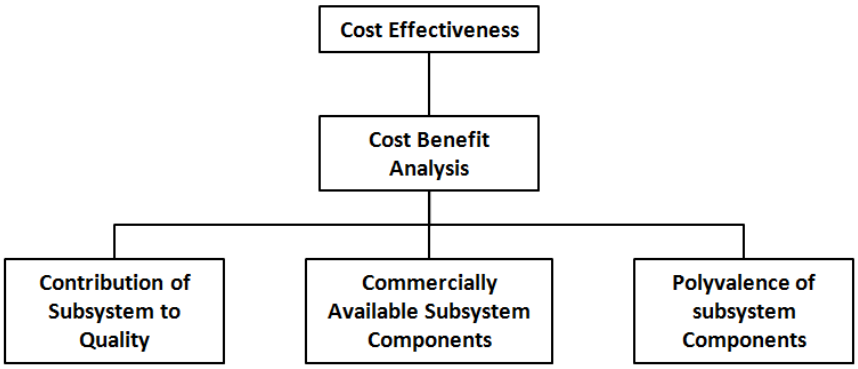


Figure 17 - Cost Effectiveness Design Sub-Objectives

Indeed, in a research environment, the purchase of equipment that can be used for the conduct of other experiments or within other projects is regarded as a benefit. For this reason, the reusability of equipment and its ability to serve multiple purposes is an evaluation factor of the cost benefit analysis that is also taken into consideration. The careful execution of the cost benefit analysis allows the spending of funds on the most useful and competitively priced equipment, thus achieving cost effectiveness in the short and long term. Three cost benefit analysis are performed in this study for the following components: the flight simulator software, the visual display and the pilot input system. A cost benefit analysis is performed for these components, because they bring the greatest contribution to the flight simulation system. The table used for comparison and for the cost-benefit analysis for each of these components can be found in Section 3.4 of this chapter, where specific flight simulator components are discussed.

3.2.3 Software Portability

From the very beginning of this study, issues of software portability were a primary concern in the design of RMC's 3DOF flight simulator. Software portability, one of the key concepts of high-level programming, is essential to create a system that can be easily adapted to other simulation hardware. Specifically, software portability refers to the features allowing the

successful reuse of code when moving software from one environment to another [15]. Indeed, it is intended that the software used for this study be easily transferable to other motion platforms in order to conduct other research work. Specifically, the software produced in this study is destined to be adapted and used for the operation of RMC's 6DOF motion platform. Software portability is closely related to cost effectiveness, because the development of portable software targeting many platforms with the same application saves time and resources. Several factors are considered to contribute to the portability of software and are used in this study to determine how to best select and implement the software components driving the flight simulator. As can be seen in Figure 18, the design sub-objectives associated with software portability are divided in three categories.

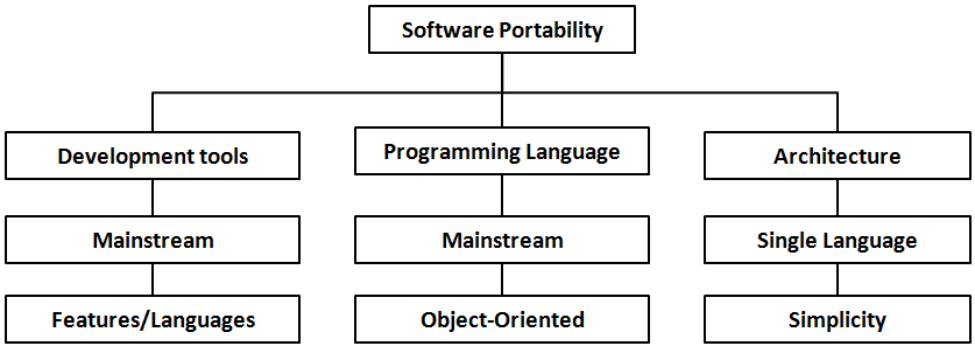


Figure 18 - Software Portability Design Sub-Objectives

First, the use of a mainstream portable programming language with a strong presence on the market is deemed very important. Indeed, the use of such a language ensures that the widest possible range of programmers can operate, maintain and modify the software. Additionally, mainstream languages are usually the ones that offer the best cross-platform compatibility. It is also preferable that the programming language selected for the implementation of the system be object-oriented. Advantages of object-oriented programming such as their modernity; abstraction and inheritance facilitate the modification and porting of software [15]. Secondly, to ensure

enhanced software portability, the selection of proper development tools is essential. As it is for programming languages, the selection of mainstream development tools allows a larger number of programmers to easily modify and port the software to other platforms. When selecting development tools, features and supported languages are factors that are taken into consideration. The best development tools usually support several programming languages and offer many interesting features to aid development. Lastly, to facilitate portability of the software components, a simple architecture must be designed. The use of a single programming language for all the software component of the flight simulator promotes simplicity. Indeed, if a system were to use several different programming languages, it reduces its portability to other platforms and creates interface problems between those languages.

3.2.4 Ease of Operation

Ease of Operation represents the ease with which users can operate the flight simulation system to accomplish their goals. This includes all phases of operation involved in a typical flight simulation run. Users interact with the flight simulation system by means of a software interface. For this reason, great care must be taken to design a GUI that promotes ease of operation. As can be seen in Figure 19, several factors must be taken into consideration to design a GUI that allows the flight simulator to be easily operated. When properly implemented, a GUI guides the actions of a user and reduces errors that could crash the system. The GUI must be designed in such a way that it does not allow the user to perform a series of actions that could lead to a crash. Additionally, the GUI must be forgiving if a user commits an error when controlling the flight simulator.

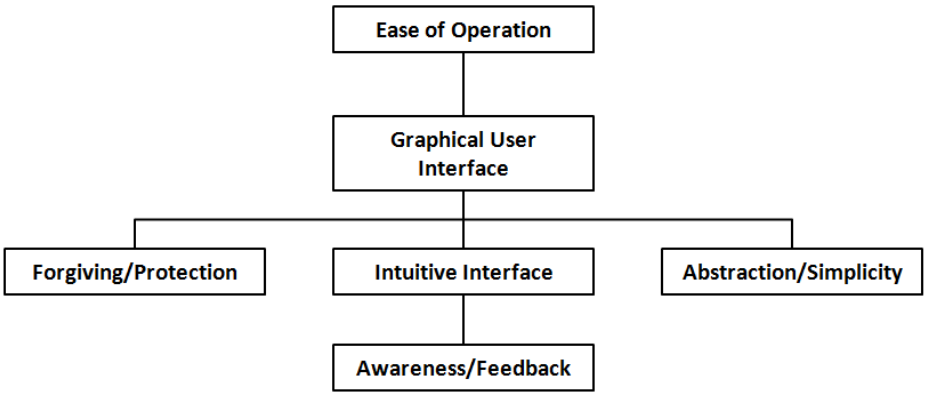


Figure 19 - Ease of Operation Design Sub-Objectives

The GUI and underlying program should never enter a state where they become "stuck" and cannot recover. Furthermore, the GUI should be intuitive and easily understood without any explanation. This includes the use of significant terms which must be used to describe the functions of each GUI controls. As a user controls the flight simulation, he must also remain aware of the system's state. For this reason, the GUI should provide feedback on the operation of program and the simulator. Finally, the GUI is an abstraction of the complicated function calls and code. Essentially, the GUI should allow for the straightforward operation of the simulator without a need to understand any of the underlying systems.

3.2.5 Maintainability

Maintainability represents the "ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [16]". Closely related to portability, this design objective is aimed at creating a software system that will be easy to maintain. As it is intended that the software produced in this study will be used by others in the future, it is important that it is properly commented and documented. Proper software documentation includes sections on: the characteristics of the

system, its architecture and design, the code, algorithms and APIs used. The specific design sub-objectives relating to maintainability can be seen in Figure 20.

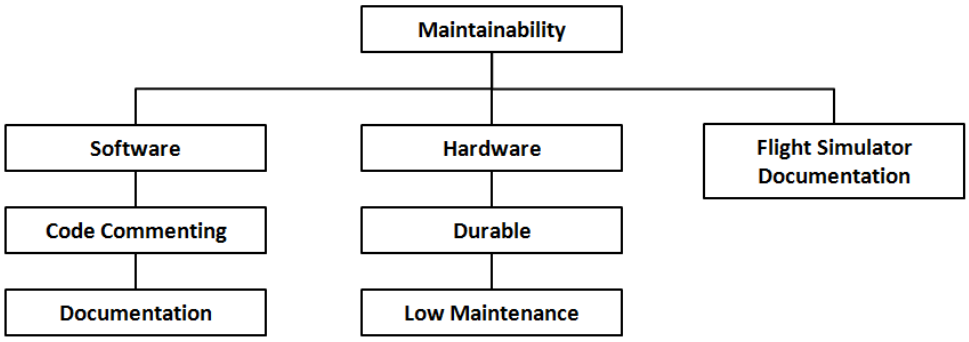


Figure 20 - Maintainability Design Sub-Objectives

A high level of maintainability allows others to adapt and evolve the software in accordance with their needs. Considerations of maintainability must also be applied to the hardware used in the design of the flight simulator. Indeed, any equipment used as part of the flight simulation hardware should be thoroughly documented and understood by the users. To promote this comprehension, a flight simulator manual should be created for the end user. Such manual should at least include sections on: the characteristics and limitations of the system, the operation of the system, the maintenance, and the safety considerations. For this reason, it is important to select equipment that comes sufficiently documented in order to increase its maintainability as part of the flight simulation system. When acquiring equipment to be used in the flight simulator, durability and low maintenance are also essential. Selecting equipment that has high maintainability increases the maintainability of the system as a whole.

3.2.6 Safety

When designing systems that interact with humans, safety is always a primary concern. RMC's 3DOF motion platform is "quite capable of generating extreme accelerations that could

damage equipment and injure personnel" [6]. As such, safety is a very important design objective. In order to ensure the safety of personnel and equipment involved with the operation of the flight simulator, both software and hardware protections are incorporated into the design of the system. Additionally, some measures are taken to ensure the safe behaviour of personnel in contact with the flight simulator. The specific design sub-objectives relating to safety can be seen in Figure 21.

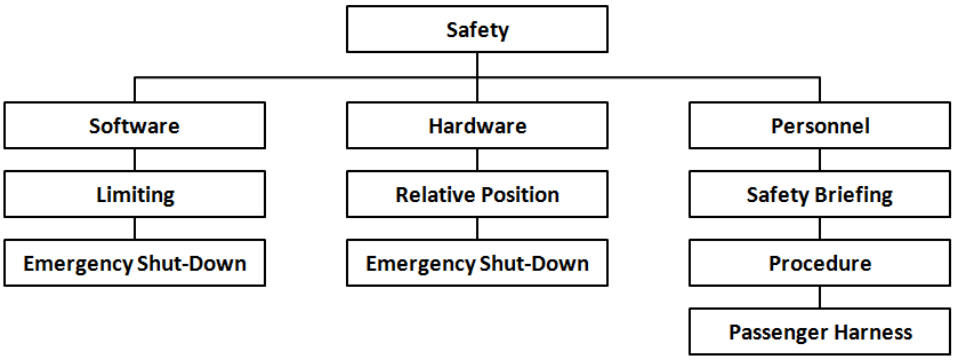


Figure 21 - Safety Design Sub-Objectives

The motion platform hardware consists of commercially available components assembled into the final system. The system has been certified to maintain its integrity provided that it is operated below certain limits. However, the system's raw capabilities far exceed the tolerance limits set by the manufacturer. Because of this, it is possible to cause damage to the platform if it is operated without safeties such as limiters. Because the platform is driven by a motion controller, the most effective way to limit the platform's velocity, acceleration and displacement is to implement software limiting. To ensure safety, software limiting can be implemented in the controller as well as in the motion cueing software. Additionally, an emergency shut-down functionality can be implemented using a simple button on the GUI used to control the simulation. If required, an emergency button can swiftly overwrite current commands and send a "stop" command to the motion platform. Despite the implementation of software safety measures,

it is also necessary to implement hardware safety to maintain control in the event of a software failure. As such, a physical emergency shut-down switch cutting power to the motion platform is necessary. This safety switch should also be in close proximity to the simulator operator to ensure a quick response. Next, the simulator should be positioned at a safe distance from potential spectators as well as the operator's station. The manufacturer suggests that a safe zone of 30 inches around the platform should be created [6]. As such, zone of operation should be clearly identified and personnel should be away from the simulator when it is in motion. To ensure maximum safety, personnel involved with the simulator should receive a security briefing as well as an outline of the emergency procedures. The security briefing is printed and placed on a wall near the operator's station. A copy of the security briefing can be found in Annex A. Finally, it is paramount to ensure the safety of any passenger in the flight simulator. For this reason, a harness system should be installed to prevent the ejection of personnel from the platform. Only through the establishment of rigid software, hardware and personnel safety measures can the risk of operating a motion platform be minimized to acceptable levels.

3.3 Constraints

The design of RMC's 3DOF flight simulator is subject to certain constraints that are inherent to the resources available as well as the equipment used for this study. Indeed, the flight simulator is designed around the existing motion platform, RMC's 3DOF, and associated hardware. This motion platform is not specifically designed to be used as part of a flight simulator and is intented to serve only general purposes. As such, it comes with certain limitations and deficiencies that must be taken into consideration during the design phase of RMC's 3DOF flight simulator. As was done for the design objectives, the design constraints are

arranged into tree diagrams. The top level tree diagram for the design constraints can be seen in Figure 22.

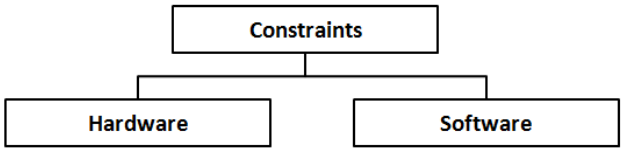


Figure 22 - Design Constraints

The constraints associated with the design of the flight simulator can be divided into hardware and software constraints. In the sections to follow, the constraints influencing the design of the flight simulator are discussed in details.

3.3.1 Hardware Constraints

The hardware constraints associated with RMC's 3DOF motion platform come from limitations of each of the different components utilized in its design. Additionally, the manufacturer, SSSI, lists constraints for the safe operation of the motion platform. Figure 23 shows the sub-level tree diagram representing the hardware design constraints.

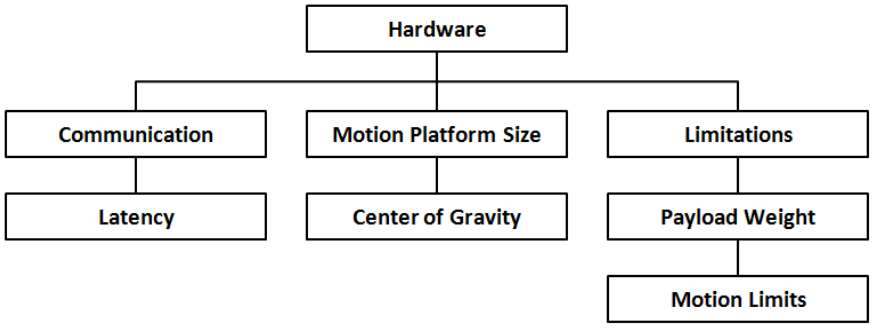


Figure 23 - Design Hardware Constraints

RMC's 3DOF motion platform is controlled by a Galil DMC 40x30 motion controller. This motion controller is widely used in various control applications and features two modes of

communication. Indeed, the motion controller can be interfaced through Ethernet or RS232 communication protocols [6]. As such, either of those two standards could be utilized to communicate with the platform. Each of these communication protocols features different speeds, which will create different latencies in the system. In this case, latency should be kept at a minimum to ensure proper synchronization and realtime operation of the motion platform with the simulation visual environment. In general, the Ethernet protocol offers much more speed and performance than the RS232 protocol. Therefore, it is better suited for the design of a flight simulator. Another hardware constraint of importance is the motion platform size. The upper platform frame measures 121.9 by 121.9 cm. Any equipment used for the flight simulator cabin and controls must remain within this area. Additionally, the center of gravity of any payload positioned on the platform must remain geometrically centered as per the manufacturer's specification. According to those specifications, the platform is designed to hold a maximum payload of 1500 lbs with its center of gravity situated 45.7 cm above the geometrical center of the platform [6]. Therefore, the passenger and the bulk of the simulator's apparatus needs to be positioned at this location and remain below the maximal payload weight.

In theory, if the motion platform respects all the aforementioned parameters and remains within its motion envelope (Table 1), it is safe to operate. However, as can be seen in Figure 24, during the conduct of this study, serious damage was caused to the motion platform during a routine session. After investigating the cause of this incident, it was concluded that SSSI's specifications were incorrect and that damage could occur even when following the prescribed limits. Indeed, when following a specific series of manoeuvres, the upper platform frame can be brought in a position where it collides with one of the motion platform drive motors. An example

of this kind of collision can be seen in Figure 25 where the motion platform is in a full pitch backward combined with a roll left.

Figure 24 - Serious Damage Sustained by RMC's 3DOF

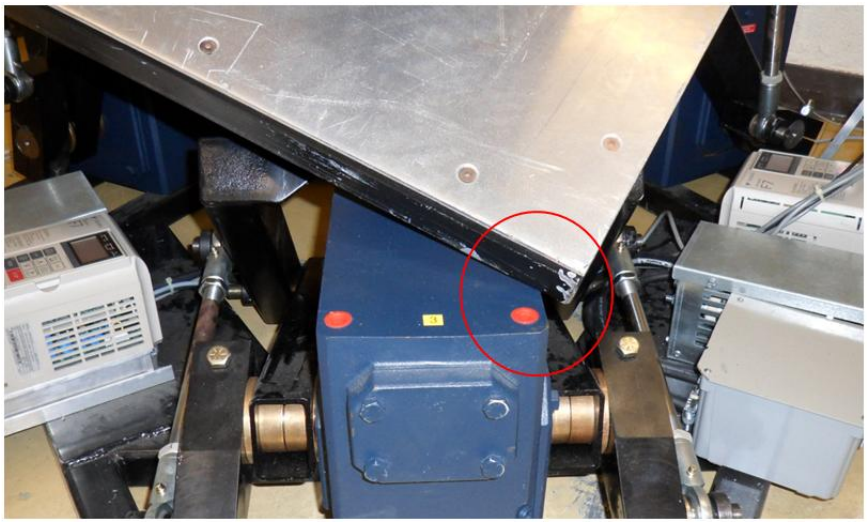


Figure 25 - Hardware Collision Caused by a Design Flaw

The possibility of a collision between the upper platform frame and two of its drive motors creates concerns for the safe operation of RMC's 3DOF. However, such collisions can be

avoided with the appropriate modifications. Through testing it was found that it is possible to prevent a rear axis drive motor collision by limiting the rear actuators three centimetres higher than originally specified or by cutting the rear corners of the platform. This can be accomplished by implementing a software limiter in the motion platform control software. Thus, the rear motion platform actuators were limited at a height of three centimetres above the drive motors. After thoroughly testing the modified software limiting, it was established that such an incident would not re-occur. This incident served as a reminder that hardware constraints must be fully understood and integrated in the design of a flight simulator. A thorough evaluation of the manufacturer's claims with regards to performance and limitations is always a good idea.

3.3.2 Software Constraints

Some of the software constraints associated with the design of RMC's 3DOF flight simulator come from the Galil motion controller provided with the motion platform. Software constraints also emerge from the requirement to select flight simulation software for the virtual environment. Additionally, software constraints emerge when a single programming language is selected in order to interface all the components of the system. Figure 26 shows the sub-level tree diagram representing the software design constraints. The Galil DMC 40x30 motion controller is built upon native C++ programming language libraries and also extends compatibility to Windows programming languages such as VB and C#. As such, either of these languages it supports could be utilized to communicate with it. Next, when selecting a flight simulation software it is important to ensure that there is a possibility to extract relevant information, such as the rotational velocities and accelerations of the aircraft, in real time by writing a plug-in.

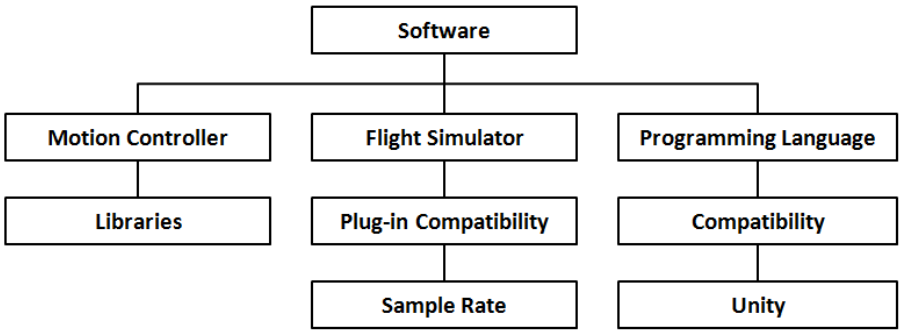


Figure 26 - Design Software Constraints

Furthermore, it is important to ensure that this information is available from the simulation environment at a sample rate that is sufficiently high to conduct realistic simulation. It is also important that a plug-in can be written in a language that can be interfaced with all the systems used in motion simulator. It is best to select a single programming language in order to write software that can interface all the systems of the flight simulator without unnecessary complications. With that in mind, the selection of a programming language is restricted to those that are compatible with both the motion controller and the flight simulator software selected to drive the simulation. When selecting a programming language, issues of portability, as described in Section 3.2.3 are also taken into consideration.

3.4 Flight Simulator Components

By carefully analyzing the design objectives and constraints, decisions can be made in order to select the appropriate, commercially available components for the flight simulation system. This section contains relevant information on the final design components that are selected based upon those criteria. More specifically, this section explains how design objectives and constraints materialized into the selection of flight simulator components. With clearly defined design objectives and knowledge of the relevant constraints, potential equipment was

evaluated, compared and selected. The design phase facilitated the selection of the appropriate commercially available components that interact together to meet the design objectives. This phase of the design is important as an understanding of the components and their interactions is the key to a successful implementation (discussed in Chapter 4).

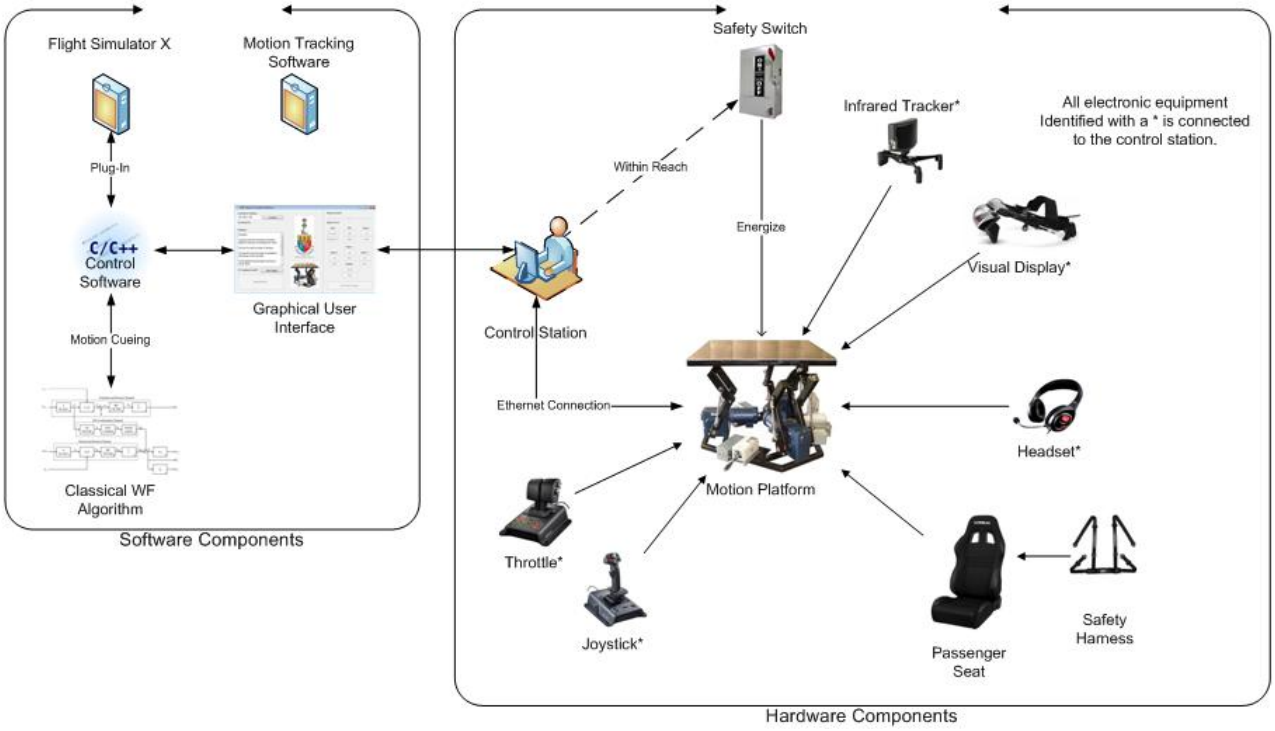


Figure 27 - Interaction of Components in the Final Flight Simulator Design

A complete and detailed listing of all the components used in the Design of RMC's 3DOF flight simulator is available in Annex B. Various hardware and software components are added to the motion platform in order to create the flight simulation system as shown in Figure 27. At the heart of the flight simulation system, a control station is the interface between the software and hardware components. A detailed wiring diagram for the hardware components is available in Annex C.

In the sections to follow, each of the key components used in the system as well as their intended role are discussed. In addition, the reasons for the selection of those components are explained in each section.

3.4.1 Simulation Environment

As stated in the design objectives, the selection of a simulation environment is an important decision in the design of a flight simulator. Three commercially available simulation environments were evaluated for RMC's 3DOF flight simulator. Microsoft Flight Simulator X (FSX) was selected as the most suitable for the design of RMC's 3DOF motion platform. As can be seen in Table 3, FSX surpasses the other flight simulation products on many of the characteristics thought relevant to this application.

Table 3 - Cost-Benefit Analysis of Flight Simulation Software

Flight Simulator	Microsoft Flight Simulator X	X-Plane 9	Flight Gear v2.0
Programming Language	C++	C++	C++
Cost	Around 50$*	Around 50$	Free
Physics Quality	Accurate	Accurate	Approximate
Graphics Quality	Very Good	Good	Average
Development Tools	Yes	Yes	No
Sample Rate	Variable (1 to 100 Hz)	25Hz	Variable
Access to Relevant Flight Data	Very Good	Poor	Poor
User Documentation	Very Good	Average	Poor

* Copies of FSX available in the department

The Flight Gear v2.0 flight simulator was taken into consideration because of its compatibility with Mathworks Simulink, a simulation environment that allows for rapid prototyping. However, after a preliminary evaluation it was determined that this flight simulation environment was not suited to drive a full motion flight simulator. Indeed, the quality of its physics is questionable. There are no development tools available and very little documentation, which makes it a very unreliable product. On the other hand, both X-Plane 9 and FSX represent acceptable flight simulation software for motion simulation. Indeed, they both deliver realistic physics of flight combined with good graphics quality. Both those products come with development tools that allow for the writing of plug-ins to communicate with the simulation environment. Microsoft's product features very good documentation and easy access to the relevant flight information. Indeed, FSX offers a complete software development kit (SDK) to easily interface with the simulation environment. This SDK (SimConnect) allows for the abstraction of many of the communications and timing issues associated with the flight simulation interfacing. This is also the case for X-Plane 9, where users can use similar libraries to easily interface with the flight simulation environment. However, X-Plane features inferior libraries without any descriptions for many of its variables. This makes it hard to know which data is relevant for the motion simulation. Furthermore, the rate at which data can be extracted from the X-Plane 9 environment (sampling rate) is fixed at 25 Hz. This fixed sampling rate does not allow for any tuning in the event that this sampling rate is insufficient. On the other hand, FSX features a variable sampling rate that allows for tuning. Overall, FSX and X-Plane 9 offer similar simulation capabilities, but FSX is more developer friendly than X-Plane 9. The implementation of RMC's 3DOF flight simulator warrants the writing of a plug-in to extract relevant information from the simulation environment. In this case, the aircraft specific

accelerations are available without the gravity vector $\vec{g}$ ($\overrightarrow{f_{AA}}$ = [a_x, a_y, a_z]) in $m \bullet s^{-2}$ and the angular rates are available in degree$\bullet s^{1}$ or radian$\bullet s^{1}$. Since the aircraft accelerations are not available with the effect of gravity, it is convenient to calculate the effect of gravity from pitch and bank information that are also available from FSX in degrees or radians. The ability to specify the sampling rate in FSX is also an important feature. For these reasons and because of its superior development capabilities, FSX is the simulation environment of choice for this study.

3.4.2 Visual Display and Sound

Several options are available for the generation of visual and audio cues for the flight simulator. However, due to hardware constraints, not all options are suitable for the design of RMC's 3DOF flight simulator. The main constraint limiting the selection of a visual display is the size of the motion platform and the requirement for maintaining any load in the vicinity of its center of gravity.

Table 4 - Cost-Benefit Analysis of the Three Major Types of Display

Visual Display	Head Mounted Display	Flat Screen(s)	Projection System
Cost	500-50000$	200-500$ Each	500-5000$
Weight on Platform	Very Light	Heavy	Heavy
Must be Secured to Platform	No	Yes	Yes
Immersive Without Closed Cabin	Yes	No	No
IR Head Tracker Compatible	Fully	Partially	Partially
Geometrically Centered	Yes	No	No
Special Requirements	No	Requires Mounting	Requires Screen

*Z800 Head Mounted Display Available in the Department.

For this reason, a convenient option is to utilize a head mounted display combined with a head motion tracker to achieve an immersive effect by providing unlimited field of regard. A comparison between the three major types of visual display system can be seen in Table 4. The use of flat screens or a projection system is incompatible with the design objectives and the constraints imposed by RMC's 3DOF motion platform. Indeed, those types of visual displays are most effective when the user is enclosed in a cockpit or a cave without outside references. Due to the motion platform size and limitations, it is not practicable to position flat screens onto the platform. Although positioning flat screens or a projection system off the platform is a possible configuration, the design of an enclosure is beyond the scope of this study and without it the immersion of the pilot would be limited.

Figure 28 - Z800 3D Visor

The head mounted display is a convenient visual display with its light weight, availability in the department and immersive effect. When combined with a head motion tracker, it becomes an acceptable display for this study. The Z800 3D visor used for RMC's 3DOF flight simulator is shown in Figure 28. The use of an infrared head tracking device allows the passenger to move his head in 6DOF to obtain an unlimited field of regard as if he is sitting inside the actual cockpit of the aircraft. This feature combined with a head mounted display is the next best thing to a cockpit replica. It is important to note that other tracking devices utilize motion sensors to

capture head movement. However, such tracking devices are not compatible with the motion platform as motion cues generated by the platform could create undesirable effects by creating false positive gyroscopic cues. Other technologies utilizing magnetic field head tracking could be viable for motion platform head tracking, but were not commercially available. The only commercially available head mounted infrared tracking system, the TrackIR 5 by NaturalPoint, is shown in Figure 29. This motion tracking system features a highly customizable motion tracking software that is compatible with FSX and other simulation software. The motion tracking software is installed on the control station where the TrackIR 5 system is connected.

Figure 29 - TrackIR 5 with TrackIR PRO Clip

To complete the immersion inside the virtual flight environment, a sound system was incorporated. Sounds can be generated by a headset or by an ambient speaker system. The generation of sounds through a headset offers many advantages for the design of the flight simulator. Indeed, the headset provides an immersive experience and most importantly, isolation from unrelated environmental noises. Isolation from unrelated noises is a very useful feature as the motion platform produces a distracting hissing noise when in operation. The Creative Fatal1ty headset (as can be seen in Figure 30), a high end commercially available headset with a microphone, was selected to generate sounds for the flight simulation. The microphone feature

can be used to allow two-way communications between the simulator's operator and the passenger.

Figure 30 - Fatal1ty Sound Headset

3.4.3 Pilot Seat and Controls

The installation of a pilot seat as well as flight controls is an essential requirement for the flight simulator. Without providing active control of the simulated aircraft to the passenger, the system remains a flight simulator, but it does not simulate the operation of an aircraft. The selection of a pilot seat is driven by the designs objectives and constraints. To create a realistic and comfortable environment, an adjustable seat suited for individuals of various heights must be installed on the motion platform. This seat should be centered on the motion platform in accordance with the manufacturer's center of gravity recommendations. In addition, a lighter seat is preferable in order to limit the forces it will generate as the motion platform accelerates. It must be fastened to the motion platform in a way that guarantees it can resist the strong accelerations of the latter. Furthermore, this seat must provide a harnessing system to effectively secure the occupant on the platform and to prevent potential ejection under high acceleration manoeuvres. Seats that are compatible with the aforementioned requirements fall under three categories: actual aircraft seat, car seats and seats specifically designed for aircraft simulators. The cost of actual aircraft seats is very high and they are difficult to obtain. Actual aircraft seats

are also likely to be bulky and quite heavy depending on their origin. The design of a custom seat for this study is not a feasible option due to the cost and unavailability of manufacturing equipment. On the other hand, car seats are readily available and come in various configurations. Designed with the safety of passengers in mind, car seats feature strong anchor points, structural integrity, seatbelt compatibility and the possibility of adjustments for the passenger. It is felt that racing type car seats are the most suitable for this application as they provide harness compatibility and protruding sides to help keep the passenger inside the seat. Most racing seats are also ruggedized with wear patches and are designed for tight spaces. Almost all commercially available racing seats boast similar features between brands. For this reason a good quality seat was selected based on availability and pricing. Weighing only 26 pounds, the Corbeau A4 racing seat was selected for the design or RMC's 3DOF flight simulator. It features solid anchor points, harness installation holes and a reclining backseat. The seat and its associated dimensions can be seen in Figure 31.

Figure 31 - Corbeau A4 Racing Seat and Dimensions

The Corbeau four point harness belt used to secure the passenger to the motion platform can be seen in Figure 32. This harness is bolted directly to the motion platform upper frame to secure the passenger on the seat. As highlighted in the design objectives, pilot controls must be

selected in order to provide realistic control of the simulated aircraft to the passenger. The realism and quality of pilot controls increases the overall realism of the flight simulation. One of the most important controls for an aircraft pilot is the control column that commands motions of pitch and roll by moving the aircraft's ailerons and elevator.

Figure 32 - Corbeau Four Points Harness Belt

Typically, two types of control column are used to control an aircraft's pitch and roll: a yoke or a joystick. The yoke is traditionally used in commercial aircraft whereas the joystick is mostly used in military or aerobatic airplanes. The most readily available type of input device to control computer based consumer flight simulator is the joystick. High end commercially available joysticks offer force-feedback to simulate the response of the aircraft's control surfaces to the aerodynamic effects on the airplane. Next, the throttle is also an important control for the aircraft pilot. It allows modifying the engine's rpm and consequently impacts the aircraft's velocity. Commercially available consumer models often feature as many as three levers that allow for the additional control of the airplane propeller pitch and engine gas mixture. With both the throttle and joystick, the pilot is using both his hands to control the aircraft. In order to control the yaw of the aircraft, the pilot must use his feet to action rudder pedals. In addition to controlling the aircraft's yaw, rudder pedals are used for ground steering as well as braking. A review of the commercially available input devices shows that most high end consumer systems feature a combination of joystick, throttle and rudder pedals assorted with numerous buttons that

can be configured to implement other functions of the aircraft cockpit. A comparison between three commercially available pilot input systems can be seen in Table 5. These pilot input systems were considered for use in the flight simulator.

Table 5 - Cost-Benefit Analysis Between three Commercially Available Pilot Input Systems

System	X52 Flight System	Pro Flight Yoke System	Flight System G940
Manufacturer	Saitek	Saitek	Logitech
Price	Around 100$*	Around 200$	Around 300$
Force Feedback	No	No	Yes
Rudder Included	Yes	No	Yes
Number of Levers on Throttle	1	3	2
Control Column	Joystick	Yoke	Joystick
Number of Buttons	23	14	32
Customizable Buttons	Yes	Yes	Yes
Hard points (for anchoring)	None	Desk Clamps	Mounting Holes

* Two sets of X52 Flight Systems are Available in the Department.

The X52 Flight System, available in the department was the first option. Despite its immediate availability, the lack of anchoring points, the absence of force feedback and the fragility of components were negative points that warranted the search for a better pilot input system. The Pro Flight Yoke System was also evaluated for potential suitability. It features a realistic yoke as well as a 3 lever throttle. Despite its many interesting features, this system comes without rudders and is provided with a clamp only designed to grip the side of a desk.

Although it does not feature force feedback, the main control column of the Pro Flight Yoke System is outfitted with a spring that creates resistance as the yoke is moved away from its neutral position.

Figure 33- Logitech's G940 Flight System

Of the system reviewed, the Flight System G940, as can be seen in Figure 33, was judged to be the best input system for this study. The G940 features high quality throttle, joystick and rudders components. Each component is sturdy and outfitted with mounting holes to allow them to be bolted to a surface. It is felt that mounting holes are very useful in this context, because they ensure the safe anchoring of the components to the motion platform. Next, the force-feedback feature of the G940 makes it one of the most realistic consumer level joysticks. The absence of force feedback or resistance in the pilot control column is a problem because it allows the pilot to easily bring the controls to their limits, which is translated into extreme motions of the simulated aircraft. This could be acceptable on a motionless flight simulator, but for a full motion flight simulator, the extremes flight motions are felt on the motion platform. Without force feedback, the passenger of the flight simulator could feel that his control inputs bring about disproportionate motion of the flight simulator. The joystick and throttle controllers come with many buttons that are fully programmable to map various secondary controls of the simulated

aircraft. Without a full cockpit this becomes a useful feature to map secondary airplane functions such as flaps and trim. The seat and controls must be installed onto the platform in a way that allows passengers of different dimensions to fit comfortably. Ideally, the seat or controls positions are adjusted to suit the preference of the pilot. Figure 34 shows a tri-dimensional concept drawing for the positioning of the pilot seat and controls.

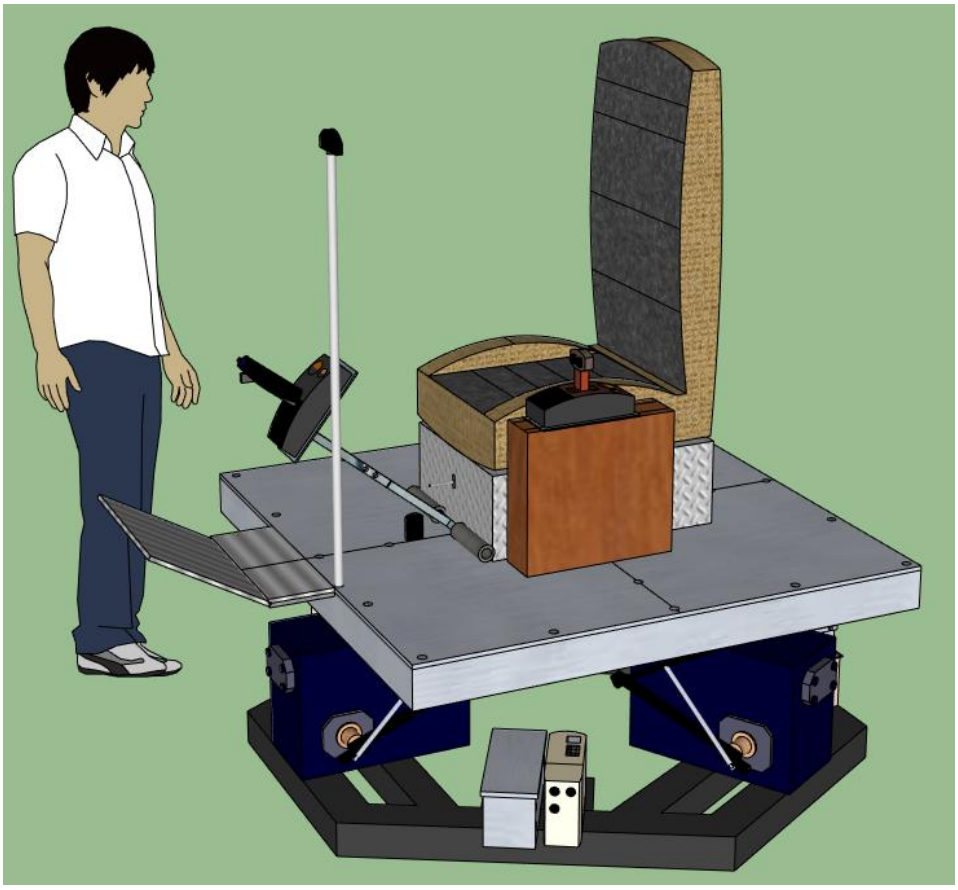


Figure 34 - Concept Drawing of the Pilot Seat and Controls

In this design, it is envisioned that the pilot seat remains in a central position to respect the center of gravity constraints. The seat can be reclined back and forth but cannot move up or down. Consequently, it is the controls position that is adjustable to best suit the pilot's preferences. The control column with the joystick on top can be lowered down to allow the pilot to board and fasten his seat belt. Once the pilot is in position, the control column is raised

between the legs of the pilot. The height of the joystick can be changed by collapsing the main control column up or down and the distance of the joystick can be modified by changing its position on the upper joystick receiving frame. The height of the throttle is fixed, but it can be moved forward and backward along a series of pre-drilled holes at the top of the support box. Figure 35 shows a close-up of the adjustments mechanisms.

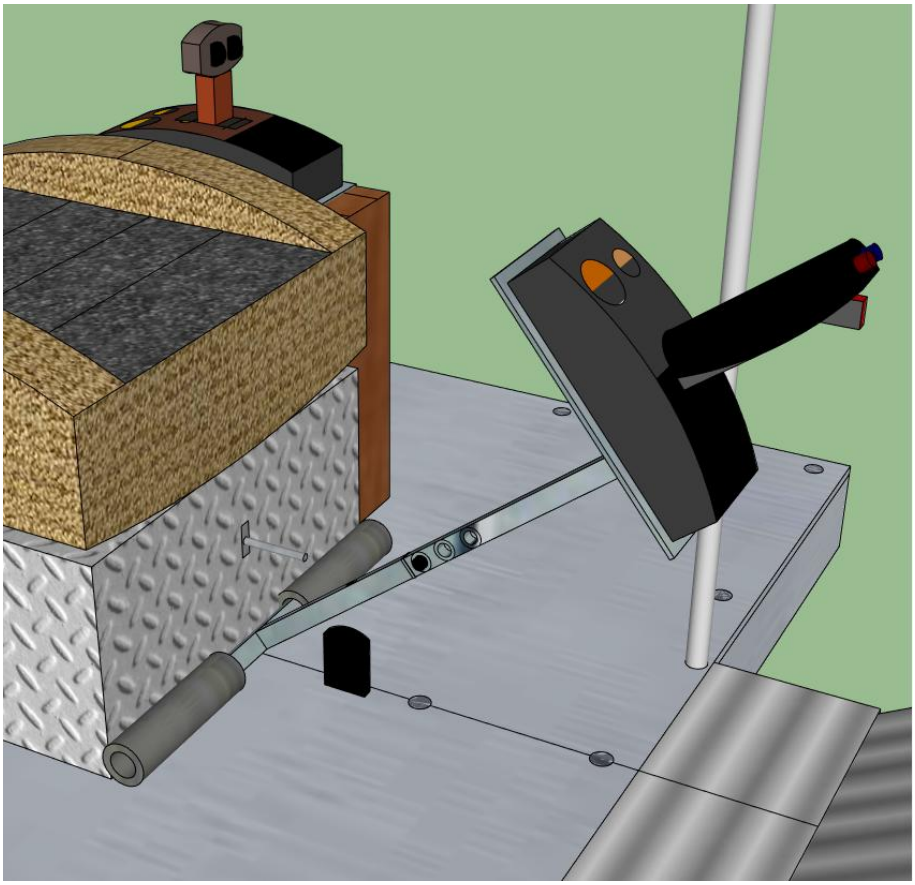


Figure 35 – Close-up on Control Adjustment Mechanisms Concept

This design could also include an extension to allow the passenger to rest his feet. However, as the motion platform is not capable of yaw motion, it is deemed that the installation of rudder pedals is not necessary. The flight simulation software can replace rudder pedals input as it allows for the configuration of automated rudder synchronization with the roll of the aircraft to produce turns. Although Figure 34 shows a potential anchoring system, it would be hard to

design a system that allows for the safe anchoring of the rudder pedals while providing adjustments. As such, rudder pedals are not part of the standard simulator configuration for this study. Nevertheless, should there be a need for rudder pedals, the dimension of the feet holder as well as mounting holes allow for a quick installation of the latter. The opinion of pilots with regards to the absence of rudder pedals can be found in Chapter 5. Finally, an adjustable height pole supporting the infrared tracker is also present on the motion platform. This pole is adjustable in order to allow passengers of various heights to sit with the tracker levelled to their head.

3.4.4 Simulator Operation and Safety

The operation of the flight simulator and safety are closely related. The simulator is designed to be operated by an operator controlling the simulation at a PC-based control station while the passenger undergoes the flight simulation. The control station is connected to the motion platform controller by an Ethernet link. Any standard computer with a network card and the appropriate control software can serve as the control station. In addition, the flight simulation and head tracking software is run on the control station. As the passenger concentrates on the actual simulation and flying of the aircraft, the operator monitors the simulation software and the motion hardware. The operator is explicitly responsible for maintaining safety by overseeing the simulation, managing any spectators and stopping the simulation if required. The operator is also responsible to assist the passenger with the boarding and tie-in procedure prior to the flight simulation. Risks to safety when using the flight simulator emerge almost exclusively from the use of the motion platform. As such, the operator must be able to quickly de-energize the motion platform when a dangerous situation arises. The surest way to guarantee absolute termination of the motion under emergency circumstances is to install a safety switch in line with the main

power supply of the motion platform. The heavy duty SquareD safety switch that was installed in close proximity to the control station can be seen in Figure 36.

Figure 36 - SquareD Heavy Duty 60A Safety Switch

Additional safety measures include the creation of an exclusion zone around the motion platform. Finally, software limiting and shutdown functionalities are implemented in the control software. Indeed, a "Stop Motion Simulation" button is implemented on the GUI of the control software to allow instant termination by the operator.

3.4.5 Graphical User Interface

The GUI is one of the most important software components in the design of RMC's 3DOF flight simulator. Using the GUI, the flight simulator operator is able to connect to the motion controller, manually control the motion platform and begin or stop the flight simulation. In addition, the operator is provided with relevant information such as the position of the motion platform actuators. To allow for the creation of graphical user interface, a GUI builder is used. Nokia's QT is a cross platform application and user interface framework. By using this C++ compatible tool, it is possible to quickly prototype and implement graphical user interfaces. The use of a GUI builder rather than programming allows the designer to focus on the layout and functionality of the GUI rather than the details of its implementation. QT also features a VS2008 add-in that allows for the seamless integration of both development tools. The final design of the

GUI can be seen in Figure 37. As can be seen from Figure 37, the GUI design allows for extensive manual control of both the 3DOF and 6DOF motion platforms.

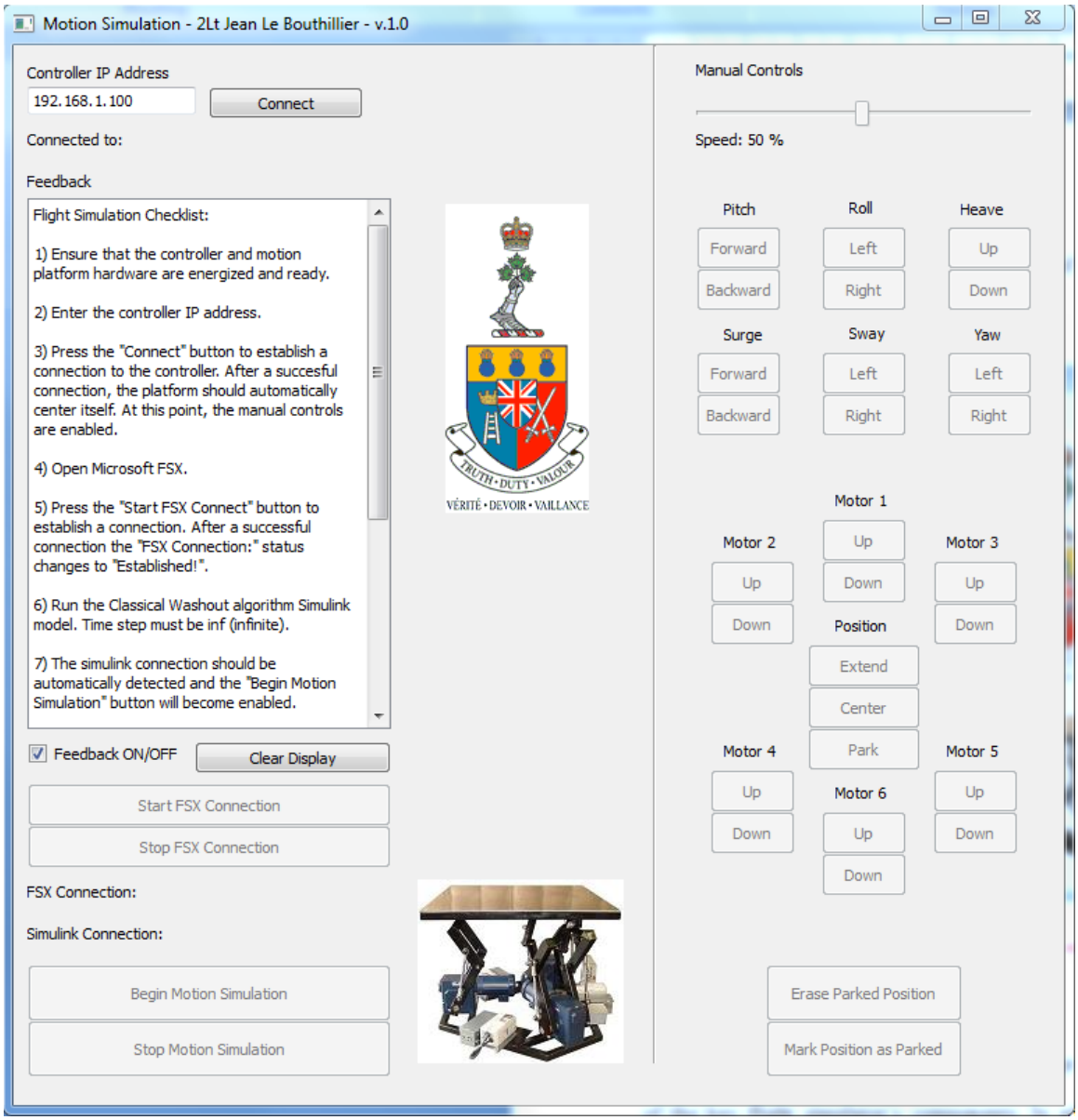


Figure 37 - Final GUI Design

A feedback window prominently displays relevant information to the user as he controls the motion hardware. Detailed instructions on how to begin a flight simulation are also included in this feedback window. The entire flight simulation setup can be controlled by four buttons

below the feedback window. The “Stop Motion Simulation” button is also the only enabled button during flight simulation in order to draw the user’s attention to it, should there be the need to hastily stop the simulation. This simple interface is highly abstracted from the complex operations required to perform the flight simulation.

3.4.6 Programming Language

The selection of a programming language is important to ensure the full integration of all of the key flight simulator’s components. In addition, the use of a mainstream programming language is conducive to a high level of portability. When looking carefully at the programming languages inherent to the key components used in the system it can be seen that C++ is the language of choice for the system.

Table 6 - Programming Languages Compatibility for Key Components

Flight Simulator Component	Programming Languages
Galil Motion Controller	C/C++/C#/VB
Microsoft Flight Simulator X	C/C++
Graphical User Interface (QT)	C/C++
Visual Studio Professional 2008	C/C++/C#/VB

As key systems such as the Galil motion controller and FSX use C++ it is imperative that other systems be integrated using the same language. The native language of key components of the flight simulation system can be seen in Table 6. In addition to providing a single language for integrating all of the key components of the flight simulator, C++ is a modern, widely used and object-oriented programming language. Those features are conducive to high a level of

portability as it can be compiled on many platforms and many programmers are likely to be able to modify the code. Finally, it can also be seen in Table 6 that Visual Studio Professional 2008 (VS2008) has been included as a flight simulator component. Although it is not a flight simulator component per se, it is included here to show its high level of compatibility towards the other components. In addition, VS2008 supports a full integration of the GUI design toolset QT. This integration is done seamlessly through the installation of an "add-in" to VS2008. Because of the quality of this integrated development environment (IDE) and its integration of GUI designer tools, it is the tool of choice to program the control software.

3.5 Summary

This chapter provided insight into the design phase of RMC's 3DOF flight simulator. In accordance with mainstream engineering design process [13], the design objectives and constraints of RMC's 3DOF flight simulator were evaluated. This chapter provided a thorough discussion of those objectives and constraints. Finally, the flight simulator's key components selected for the design of RMC's 3DOF flight simulator were discussed. Having completed a thorough design process and identifying the components required for the design, the next chapter will discuss the implementation of the flight simulator.

CHAPTER 4 – IMPLEMENTATION

4.1 Introduction

After the analysis of the design objectives and constraints, and having selected the system's components, the implementation of the system can begin. In the implementation phase, the software components are created in accordance with the conclusions drawn in the design phase. The most significant part of the implementation in this study is the software implementation. The software implementation follows a methodology that ensures the design is respected. The installation of the hardware components onto the motion platform is also undertaken during the implementation phase.

In this chapter, the implementation of the software components required to enable flight simulation will be discussed. More specifically, the implementation process and decisions taken during this process will be explained. The intent is to provide the reader with an understanding of the work conducted during the software implementation phase. An explanation of the implementation philosophy will be provided as well as a discussion on the implementation of specific components of the system. Next, the networking elements used in the flight simulation system will be explained to further expand the reader's understanding of the system. References to the annexes where, in some cases, more specific implementation details are shown will also be given throughout this chapter. Then, the installation of physical components onto the motion

platform will be detailed. Finally, the validation of both the software and hardware implementation by means of technical and user reliant testing will be discussed.

4.2 Software Implementation

The software implementation of the flight simulation system encompasses four main software components and utilizes both UDP and TCP networking protocols to establish communication between certain components.

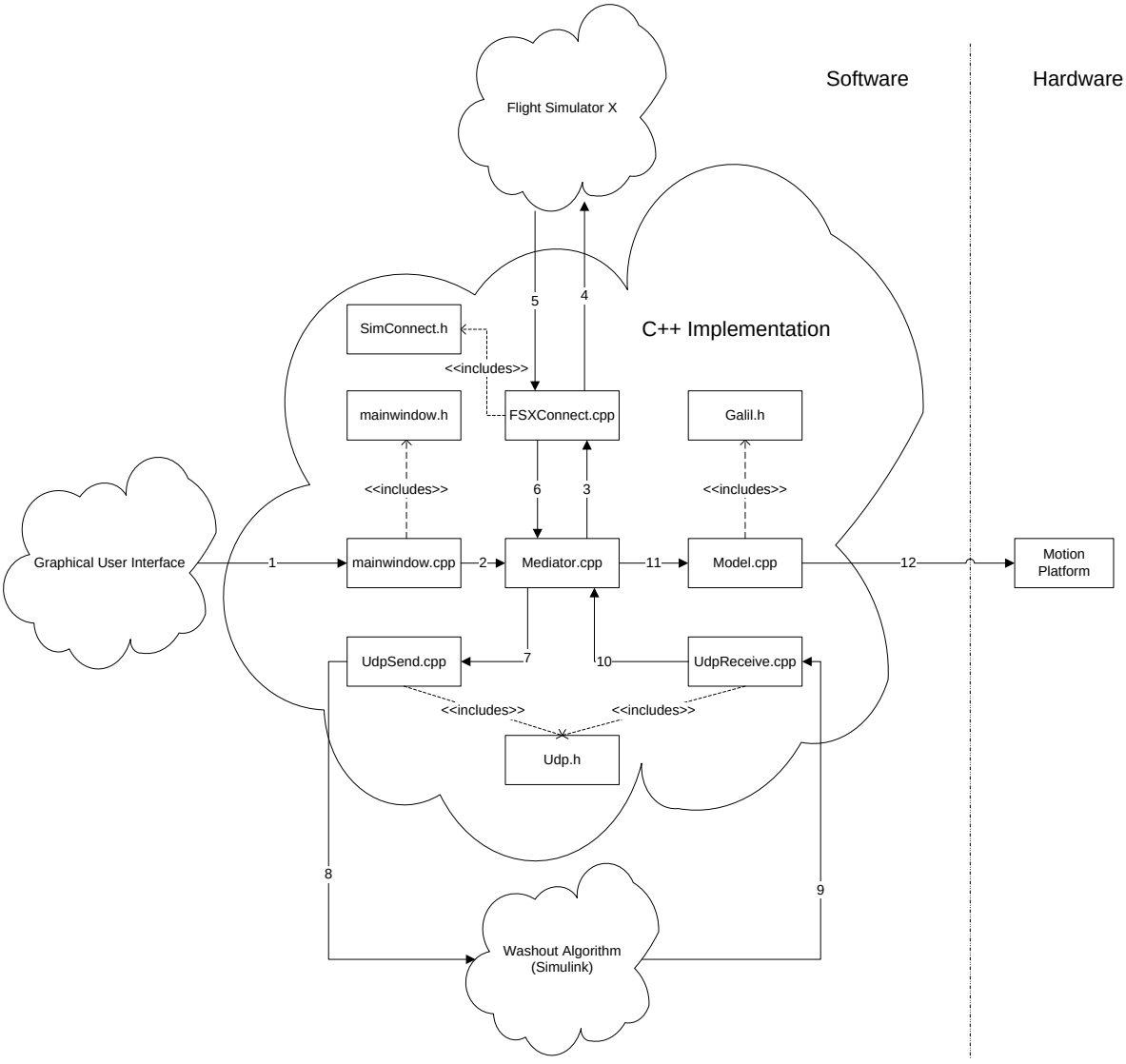


Figure 38 - General Interaction of Software Components and Flow of Information

The four main software components are the GUI, Flight Simulator X, the classical WF algorithm and the C++ implementation. Together, these software components form the flight simulation system. In Figure 38, the general interaction of the four main software components can be seen. In addition, the typical flow of information from the GUI to the motion platform is depicted. In essence, the information travels from the GUI to the C++ implementation that is the core of the system. From there, each component can be reached by the C++ implementation, which is able to bring them together into a cohesive system that eventually sends the proper commands to the motion platform. Figure 38 provides a top level general picture of the software implementation that will be discussed in this chapter. Specific interactions, relationships and networking between each component will be thoroughly discussed in the sections to follow or in the annexes.

4.2.1 Purpose and Use Cases

In order to properly conduct the software implementation phase, it is important to understand the purpose and use cases of the system. The goal here is not to go into any details of the implementation or the software architecture, but to examine the fundamental tasks that the users of the system are trying to accomplish. After having clearly identified those “use cases”, it is easier to implement a solution. The purpose of the software in this research is to allow the simulator operator to interact with the motion platform in order to conduct flight simulation. To empower the simulator operator to do this, the software must provide several key functionalities. Use case diagrams are commonly used to illustrate the functionalities of a system in terms of its users [17]. In addition, use case diagrams are a part of the Unified Modelling Language (UML) toolbox for the design and visualization of object-oriented software [17]. The granularity or level of detail of use case diagrams is generally left to the discretion of the author. However, it is

preferable to keep those diagrams at a low level of granularity to ensure that no architecture details are suggested by the diagrams. In use case diagrams the actors or users are represented by stick man figure and the actions that the user must be able to accomplish are illustrated by ovals. One such use case diagram can be seen in Figure 39 where the three fundamental use cases for the system can be seen.

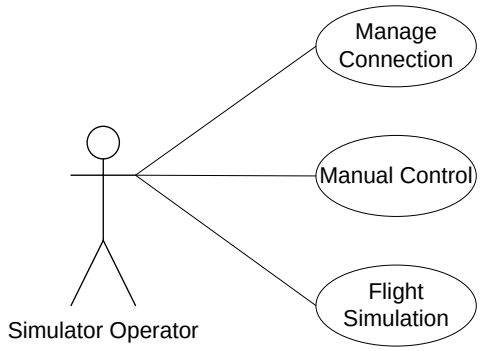


Figure 39 - System Use Case Diagram

First, the simulator operator must be able to use the software to manage a connection with the motion platform. More precisely, this includes the ability to connect and disconnect from the motion platform. Then, the software must allow the simulator operator to manually control the motion platform. This includes controlling the speed of the motion platform and moving it in any possible position. The third functionality that must be provided by the software is to allow the simulator operator to conduct flight simulation. This includes the initiation of a flight simulation and its termination from the software GUI. The software implementation is conducted while keeping the use cases in mind and is complete when the simulator operator is able to conduct the use cases with ease.

4.2.2 Functional Approach to Implementation

As stated in the design objectives of Chapter 3, it is important to design software components that are simple to use, maintain and that are portable to other applications such as the control of RMC's 6DOF motion platform. As such, a functional approach has been taken for the implementation of the software components in this study. A functional approach is the term utilized here to describe the implementation of the required software components in a way that ensures that each software components of the system serves a unique and independent purpose from the others. The functional approach described here is in line with object-oriented best practices [18]. The idea is to separate the software into several simpler parts (components) to accomplish each of the main functionality of the system. In addition, the goal is to implement the system in a way that separates software components from each other as much as possible. As opposed to implementing all the main functionality of the system together, they are separated into different software components that only perform specific and unique actions. To accomplish this, the C++ object-oriented programming language is very useful as it is already designed to facilitate this kind of approach in the implementation. This approach provides advantages later in the software life cycle as it reduces dependencies and coupling between each software components. When there is a need to modify a specific component or to add functionality, the idea is that only the required components are modified or new ones added without the need to modify the entire system. Figure 40 illustrates the difference between a coupled approach and the functional approach used in this study. In the coupled approach, we can see that four software functionalities are implemented as a single software component.

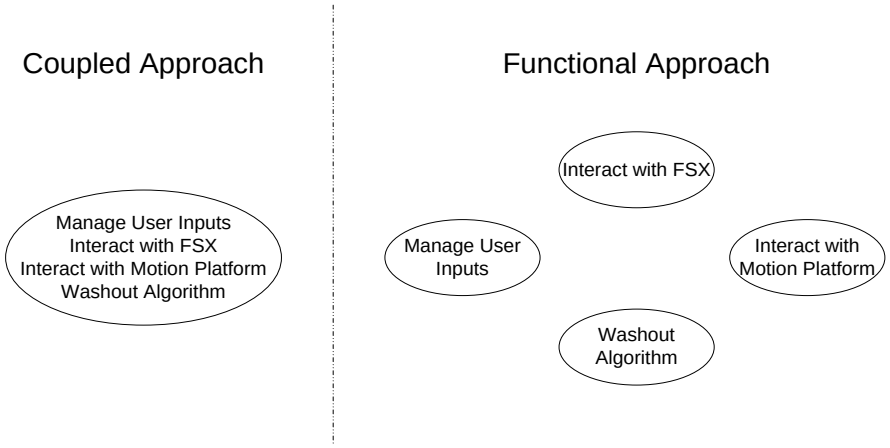


Figure 40 - Comparison between Coupled and Functional Approach

This makes such a system very resistant to change or the addition of new functionalities. In the functional approach, four software components are created for each of the four distinct software functionalities. As a result, each of these software components are now independent from each other and change can be made on any of the components without affecting the others. For example, if we wanted to change the flight simulation environment to X-Plane 9 instead of FSX, we would only need to change the "Interact with FSX" software component. This change would be effected with minimal or no disturbance at all to the other software components. Finally, since the software is now divided into several software components, it is necessary to implement an effective way for each software components to interact with each other. As will be seen in the next section, the use of design patterns is an effective way to solve the communication problem caused by the separation of software functionalities into different software components.

4.2.3 Mediator Design Pattern

When attempting to design software components that are reusable and easily maintained, it is important to ensure that there are no dependencies among them. This can be easily

accomplished by separating the software components in accordance with their respective functionality. In order to control this level of abstraction between software components, it is necessary to use a software design pattern that is tailored for this kind of application where a "many to many" relationship naturally occurs between the system's software components. The Mediator design pattern is well suited for this application because it abstracts the behaviour of the system's software components from each other [19]. In other words, when software components need to work together, they must all go through the mediator that handles the components interactions. The mediator is the only software component to know and understand the behaviour of all of the system's software components. The main advantage of this design pattern is that the software components or objects (in object-oriented programming terminology) do not require any knowledge of the other objects of the system. They only need to be able to interact with the mediator.

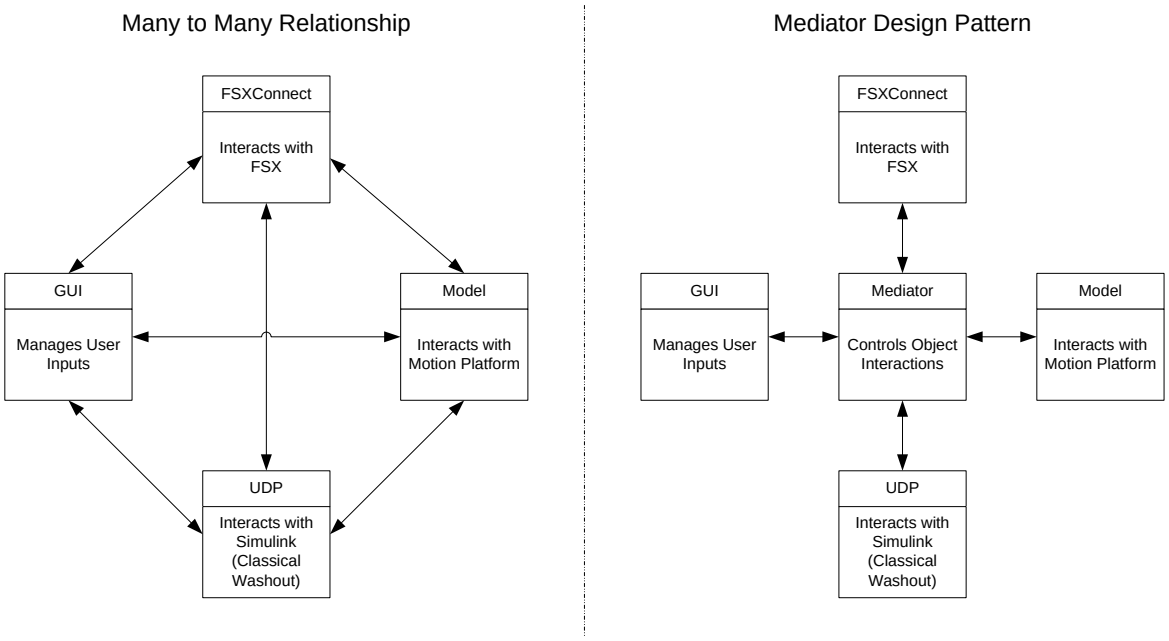


Figure 41 - The "Many to Many" Relationship and the Mediator Design Pattern

As a result, the software components are uncoupled from each other and a modification to any of the components in the system does not affect the others [19]. As can be seen in Figure 41, the Mediator is an object that defines and controls how the objects of the system in a “many to many” relationship interact with each other. For example, if a user commands a motion by clicking a button on the GUI, this command is passed to the mediator which then forwards it to the model. The model has specific knowledge on the motion platform and is responsible for communicating with the motion platform and executing commands. Hence, once the model receives a command from the mediator, it simply executes it without any knowledge or care for its origin. Then, the model returns information on the failure or success of the command to the mediator which sends it back to the GUI, which is also unaware of the true origin of this information. The use of a model in this implementation enables the GUI or any other components to send motion commands without any knowledge of the communication protocols required to communicate with the motion platform. Because of this abstraction, it is very easy to maintain and change the system with minimal effort. The same condition exists for each of the four objects depicted in Figure 41 as they all benefit from the abstraction layer created by the Mediator design pattern. A complete UML class diagram of the software implementation using the Mediator design pattern can be found in Annex D. The complete C++ code can also be found in Annex K.

4.2.4 Flight Simulator X Plug-in

To achieve a realistic simulation, it is essential to provide the flight simulator passenger with high quality visual and audio inputs. In addition, it is even more important to generate aircraft translational accelerations and angular rates that are realistic and can be used to calculate the motion platform displacements. After having decided that FSX is the flight simulation

environment of choice for this study, it is necessary to find a way to extract the translational accelerations and angular rates in order to send them to the classical WF algorithm for motion cueing calculations. Fortunately, FSX comes with a powerful client-server communications protocol (SimConnect) that allows accessing the internal data of FSX in an asynchronous manner. The protocol is considered asynchronous because a publish/subscribe method is used to establish the communication between the client and server [20]. As a result, once a client subscribes to a data or event, the server internally processes and returns the information at a frequency that can vary and is specified in the initial client request. Since the requests are internally processed by the server and never handed for processing to the client, this communication is considered to be asynchronous [20].

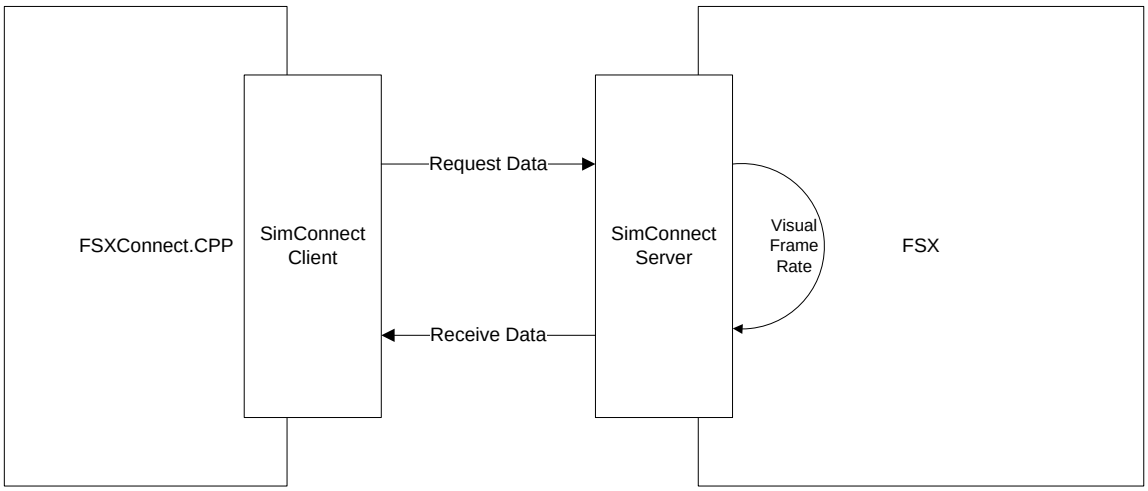


Figure 42 - SimConnect TCP/IP Communications Model

As can be seen in Figure 42, the TCP/IP Winsock networking protocol is used to establish network communication between a SimConnect client on top of an external (out-of-process) executable and the SimConnect server (in-process component of FSX). In this study, a C++ object, FSXConnect.cpp, was implemented to establish a SimConnect communication with FSX and to obtain the relevant data and events. It is possible to retrieve data from FSX in three

different frequency modes: every visual frame, every simulation step or every second (1 Hz) [21]. In this study, it was decided to use the visual frame rate as the polling frequency for the data. The reason for this is that it provides better control due to the synchronization of visual frames with motion frames. Indeed, it is not possible to control the exact frequency when using simulation steps and the 1 Hz frequency is simply too low to produce a high quality motion cueing.

Table 7 - Information Requested from FSX Using FSXConnect

Name	Type	Function	Unit
ACCELERATION BODY X	Data	*x* axis acceleration	$m \bullet s^{-2}$
ACCELERATION BODY Y	Data	*y* axis acceleration	$m \bullet s^{-2}$
ACCELERATION BODY Z	Data	*z* axis acceleration	$m \bullet s^{-2}$
ROTATION VELOCITY BODY X	Data	*p* angular velocity	$deg \bullet s^{-1}$
ROTATION VELOCITY BODY Y	Data	*q* angular velocity	$deg \bullet s^{-1}$
ROTATION VELOCITY BODY Z	Data	*r* angular velocity	$deg \bullet s^{-1}$
PLANE PITCH DEGREES	Data	Plane pitch degree	*deg*
PLANE BANK DEGREES	Data	Plane bank degree	*deg*
EVENT_SIM_PAUSE	Event	Pause event	1 or 0

When using the visual frame rate, there is always and exactly one motion command sent per visual frame. Moreover, the visual frame rate also allows for the simple modification of the motion simulation rate by modifying the visual frame rate directly through the in-game FSX settings. The information requested from FSX through FSXConnect.cpp can be seen in Table 7. Essentially, the bulk of the data requested from FSX is the data required by the classical WF algorithm (translational accelerations, pitch, bank and angular rates). The pitch and bank information are required to calculate the effect of gravity on the aircraft for the translational acceleration channel. It is necessary to collect this information to address the fact that FSX

accelerations returned for each axis of the body do not include the effects of gravity. As such, the individual acceleration on each axis due to the force of gravity is calculated using Equations 2.12 to 2.14 from pitch and bank information obtained in FSX where α is the pitch angle and β is the bank angle. This formulation for *y* axis acceleration is however not proper for simulation as it does not account for the centrifugal force which is not available in FSX and generally brings the *y* axis acceleration to zero in a coordinated turn. If this simplified formulation would be used, unrealistic flight simulator tilt would occur during the simulation of coordinated turns. Due to this limitation, the *y* axis acceleration due to gravity is set to zero for this study. This zeroing of g_y is appropriate as it will level the platform during a turn (which is what is supposed to happen as the centrifugal force would neutralize the *y* axis acceleration).

$$g_x = \sin(\alpha) * 9.81\ m \cdot s^{-2} \tag{4.1}$$

$$g_y = \sin(\beta) * 9.81\ m \cdot s^{-2} \tag{4.2}$$

$$g_z = [\cos(\alpha) * 9.81\ m \cdot s^{-2}] * \cos(\beta) \tag{4.3}$$

Then, the pause simulation event is requested in order to know when the simulation is paused or running. This information is used to know when the simulation is paused in order to set all outbound FSXConnect.cpp data to zero and ensure that the motion platform stops moving. It is important to highlight the fact that data is only received from FSX when it has been modified in the flight simulation. In other words, if there is constant x axis acceleration, the acceleration value will be published only once. The next publish by the SimConnect server will only occur when the acceleration value has changed in the simulation. The differences between the axis systems used by the classical WF algorithm and FSX can be seen in Figure 43. It is

important to note that the axis orientation and axis rotation is different between the two environments.

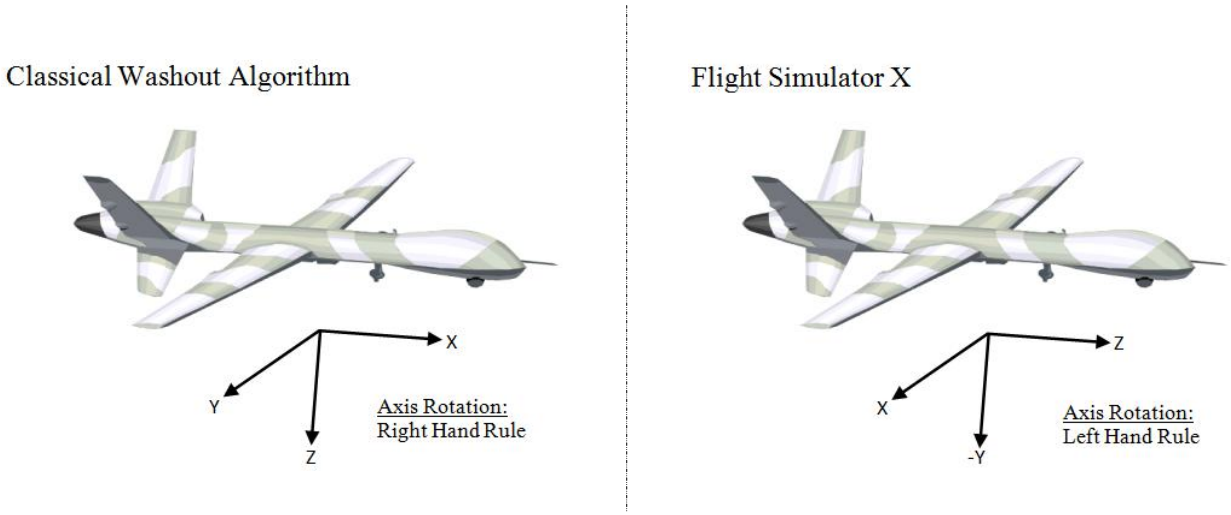


Figure 43 - Different Axis Systems between Classical WF Algorithm and FSX

Thus, the FSX axis system is converted from FSX to classical WF algorithm axis system inside FSXConnect.cpp. No officially published data could be found on the FSX axis system. As a result, it was necessary to extrapolate the axis system by conducting an experiment where specific manoeuvres known to generate motion along or about a specific axis were performed to uncover the orientation and axis rotation of each axis. More information on the specific C++ implementation of FSXConnect.cpp can be found in Annex K.

4.2.5 Classical WF Algorithm Implementation

The classical WF algorithm can be challenging to implement as it contains a fair amount of mathematical operations such as continuous time integrations, derivations and matrix multiplications. Reid and Nahon demonstrated that it is feasible to implement the classical WF algorithm using traditional programming languages such a FORTRAN [8]. They also highlighted the difficulties associated with the conduct of continuous time integration and derivations with

an acceptable level of mathematical precision while keeping the processing times very low [8]. Indeed, they highlighted the importance of fast and accurate integration methods to maintain an overall low latency in the flight simulation system [8]. More than 25 years after the ground-breaking work of Reid and Nahon [8, 9 and 10], new technologies such as Mathwork's Simulink have emerged, and make it easier to deal with concerns of accuracy and processing time when performing mathematical operations. "Simulink is an environment for multi-domain simulation and Model-Based Design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let you design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing" [22]. The use of Simulink allows focusing on the actual time-varying system instead of the details of implementing mathematical operations. Simulink also leverages all the power of Mathwork's MATLAB, a high level programming language and environment, and is able to perform almost any mathematical operations faster than traditional programming languages [23]. It also allows developers to quickly implement and test algorithms in a drag and drop fashion. For these reasons and because of its availability to students at RMC, it was selected for the implementation of the classical WF algorithm in this study. Figure 44 shows the implementation of the classical WF algorithm in the Simulink environment. In Figure 44, the top level view of the classical WF algorithm can be seen where the channels and other key components are denoted using different colors. It includes the main components of the algorithm as well as some UDP networking components that will be discussed in the next section. Simulink allows for a clean layout of the top level algorithm blocks while hiding many layers of underlying complexity. Each Simulink block can be opened with a double click to reveal its internal components. This is a useful feature which allows the designer

to quickly navigate from the top level of the algorithm all the way down to the details of the implementation.

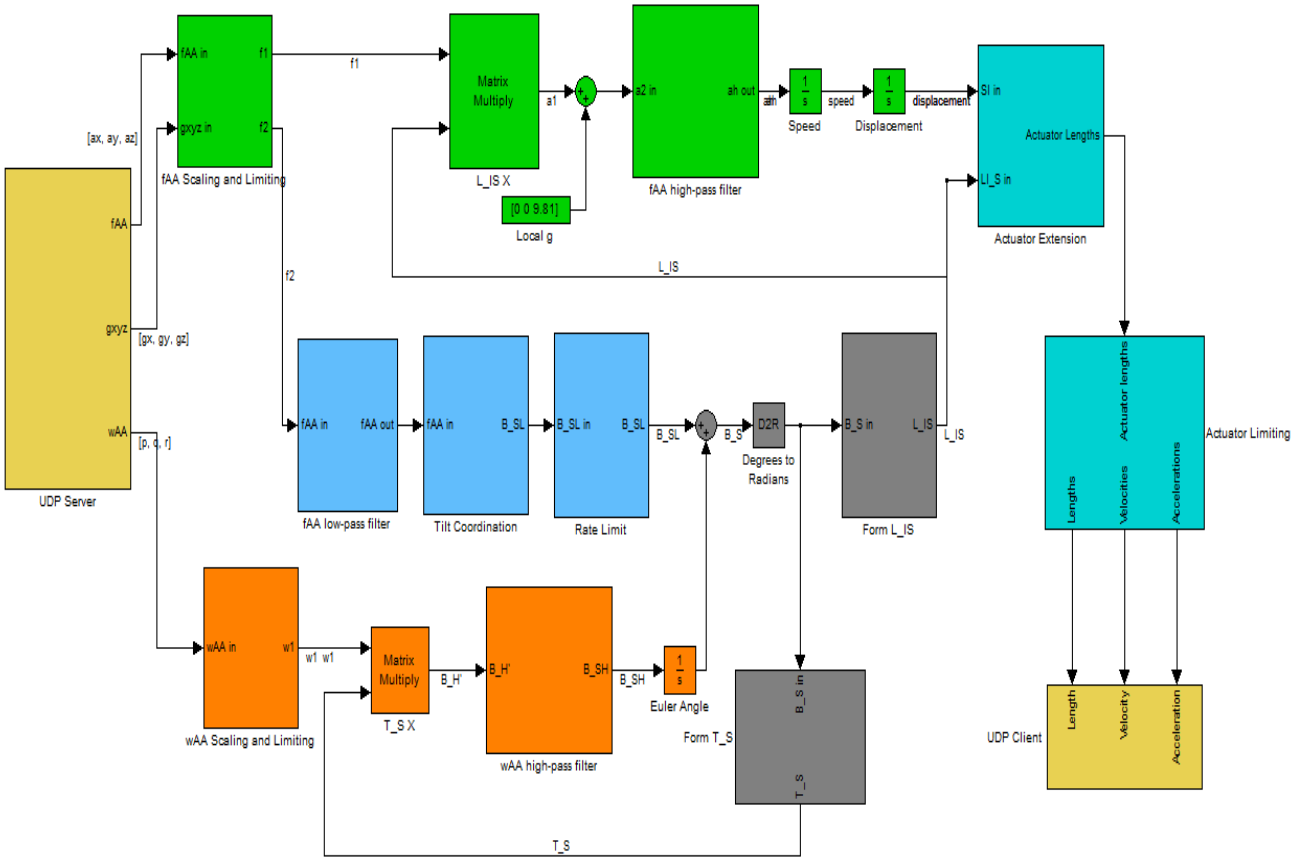


Figure 44 - Simulink Implementation of the Classical WF Algorithm

Figure 45 shows the internal components of a low pass filter block, which would be revealed by a double click of the block in Simulink. It is important to specify that the implementation of the classical WF algorithm in Simulink required special MATLAB and Simulink add-in packages that are called “toolboxes and blocksets”. It was necessary to acquire the “Instrument Control Toolbox”, “Control System Toolbox” and the “Signal Processing Toolbox and Blockset” in order to implement the algorithm. These additional packages were required to implement low and high pass filters and to establish network connectivity to and from the Simulink environment.

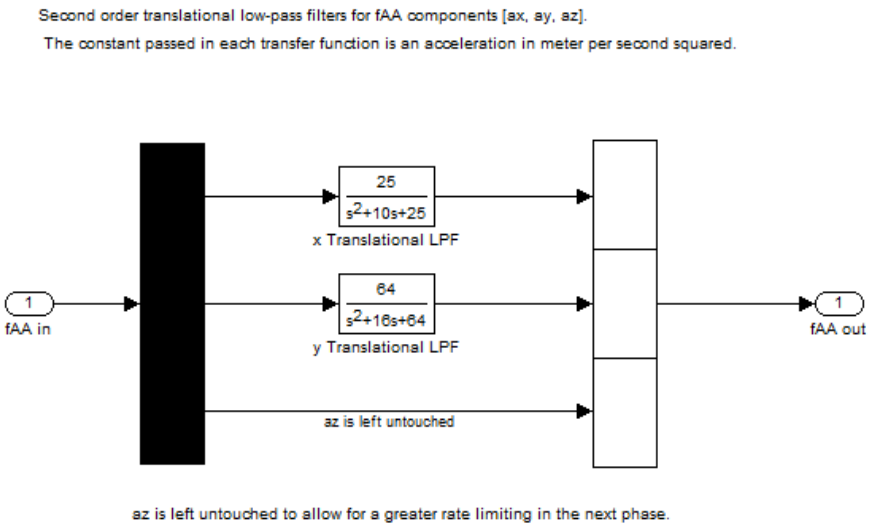

Second order translational low-pass filters for fAA components [ax, ay, az].
The constant passed in each transfer function is an acceleration in meter per second squared.
25
s2+10s+25
x Translational LPF
64
s2+16s+64
y Translational LPF
az is left untouched
1
fAA in
1
fAA out
az is left untouched to allow for a greater rate limiting in the next phase.

Figure 45 - Internal Simulink Block View Sample

Some minor configuration problems relating mostly to the complexity of the environment were encountered during the implementation, but these problems were eventually resolved after reading the Simulink help files, which are very thorough. The Simulink environment allows developers to run several iterations of a model as fast as the host computer will allow it. However, in the case of a real-time system such as a motion platform, it is necessary to "slow" down the simulation in order to achieve real-time speed. Although this is not a feature that is available using the standard version of Simulink, it is possible to slow down the simulation by comparing the host computer's clock to the simulation's clock and by putting the simulation to "sleep" as required in order to achieve real-time speed. Such tweaking of the Simulink environment is more advanced, but is considered a mainstream practice amongst the MATLAB community. As such, several blocks and code snippets are available on Mathwork's website to achieve real-time speed. One such block, called "RTsync" was downloaded and is used to control the simulation's speed.

2.4.4 Calculating the Actuator Extensions, Velocities and Accelerations

A technique for calculating the actuator extensions is discussed in [8]. However this technique was found to be inadequate for this study and for use with RMC's 3DOF motion platform. Actuator extension calculation techniques depend on the type of hardware used and vary depending on the type of commands required by the motion platform. The work conducted by [8] to solve this problem does not apply well to RMC's 3DOF motion platform. Hence, a different technique inspired from [8] is used in this study for RMC's 3DOF.

Using the simulator displacement $\overrightarrow{S_I}$ from the translational motion channel (in this case only in the z axis) and the Euler rotation matrix from the tilt-coordination and rotational motion channel, it is possible to calculate the position of the upper motion platform frame.

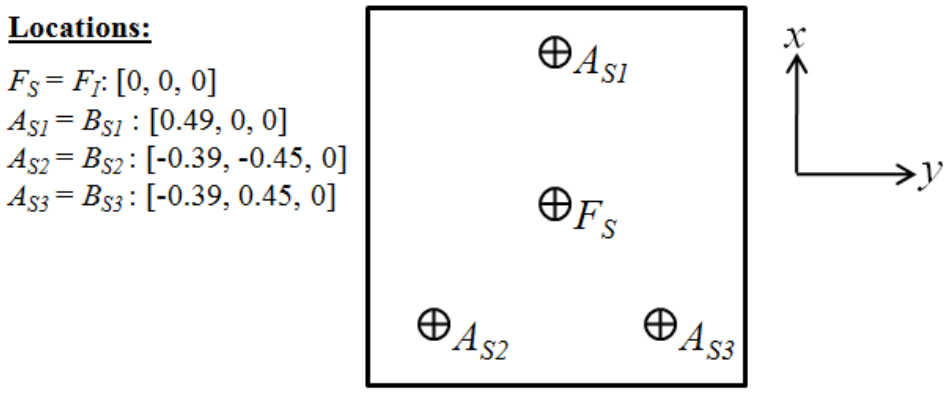


Figure 46 - Geometrical Layout of the Actuators (Top Frame View)

Figure 46 shows the geometrical layout of the motion platform base where the positions of the actuator attachments are denoted in by $\overrightarrow{A_{S_I}}$ and represented in space using the Cartesian system in meters. In the case of RMC's 3DOF, the positions of the actuator attachments on the upper motion platform frame, denoted by $\overrightarrow{B_{S_I}}$ are the same as for the motion platform base. Knowing the geometrical layout of the motion platform allows for the superimposition of the

upper and base platform frames in order to derive the actuator's lengths. In Figure 47, an example of such a superimposition can be seen where the upper motion platform frame is undergoing a pitch backwards combined with a heave up.

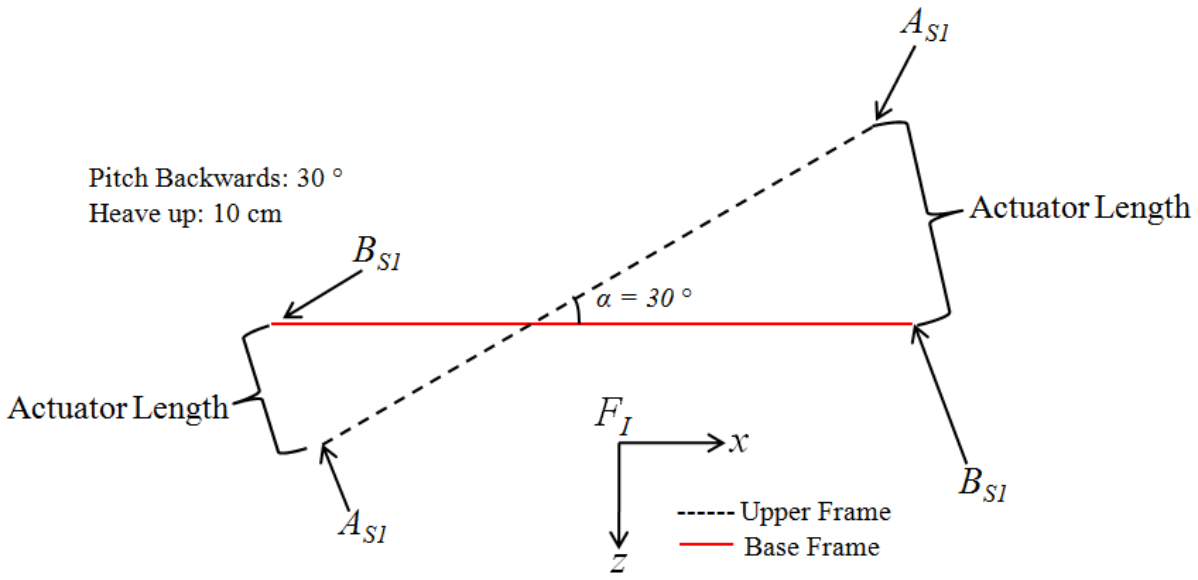


Figure 47 - Superimposition to Calculate Actuator Extensions

This superimposition technique allows for the calculation of actuator's extensions l_i using Equation 4.4 where the Euclidian norm for the difference in position of the actuator attachments between the upper and base platform frames is taken. The polarity of the z axis is kept for the z axis coordinate with the sign operation (sgn_z). It is necessary to use the sign operation since the sign is lost when taking the Euclidian norm to get the length.

$$l_i = sgn_z\left[\left(L_{IS} \cdot \overrightarrow{A_{is}} + \overrightarrow{S_I}\right) - \overrightarrow{B_{IS}}\right] * \left\|\left(L_{IS} \cdot \overrightarrow{A_{is}} + \overrightarrow{S_I}\right) - \overrightarrow{B_{IS}}\right\| \tag{4.4}$$

The washout position of the motion platform is reached when the actuators are extended to half their maximum elongation at what is called the central position. In other words, the motion platform will always seek to return to this position. For RMC's 3DOF, this leaves an available distance of 0.20 meters for each actuator to move either up or down. It is from this

position that the lengths calculated from 4.4 are applied. As can be seen from Equation 4.5 and 4.6, the velocities v_i and accelerations a_i are derived from the lengths using its first and second derivative respectively. Finally, the lengths, velocities and accelerations are limited according to SSSI's specification (Table 1) using a limiter as described in Equation 2.8.

$$v_i(t) = \frac{dx}{dt} \tag{4.5}$$

$$a_i(t) = \frac{dv}{dt} \tag{4.6}$$

4.2.6 Networking

The implementation of the flight simulation system relies on the use of networking for communications with FSX and Simulink. In addition, network communications are used to communicate from the C++ implementation to the Galil motion controller. In Figure 48, we can see that both UDP and TCP/IP Networking protocols are present in the flight simulation system.

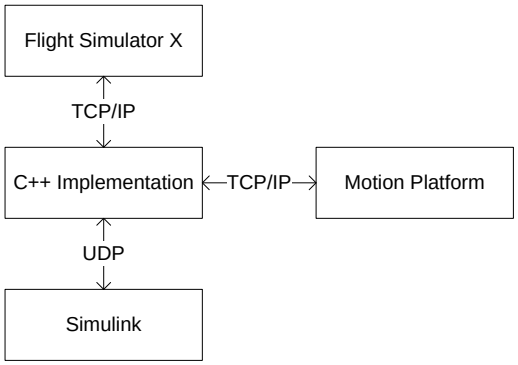


Figure 48 - Networking in the Flight Simulation System

The TCP/IP networking segment between the C++ implementation, FSX and the motion platform controller are easily handled as they rely on third party libraries that take care of the networking communications. Indeed, when using SimConnect to connect to FSX, the TCP/IP networking details are abstracted and operate without the knowledge of the user. Similarly, when

communicating with the motion platform, the Galil C++ library that is provided with the motion controller handles the networking communications. On the other hand, the UDP communication protocol used to communicate to and from Simulink had to be implemented in C++ using the Windows Sockets API (Winsock). The C++ UDP implementation allows sending and receiving double values over UDP. As can be seen in Figure 49, there are three C++ classes that had to be used to allow for the transfer of those double values.

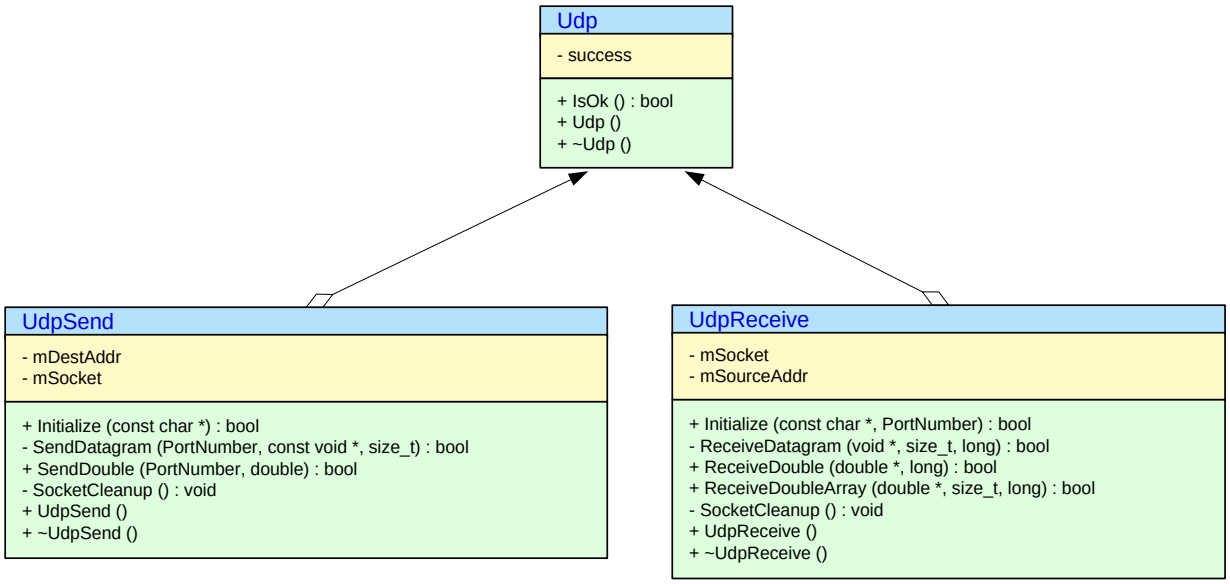

Udp
- success
+ IsOk () : bool
+ Udp ()
+ ~Udp ()
UdpSend
- mDestAddr
- mSocket
+ Initialize (const char *) : bool
- SendDatagram (PortNumber, const void *, size_t) : bool
+ SendDouble (PortNumber, double) : bool
- SocketCleanup () : void
+ UdpSend ()
+ ~UdpSend ()
UdpReceive
- mSocket
- mSourceAddr
+ Initialize (const char *, PortNumber) : bool
- ReceiveDatagram (void *, size_t, long) : bool
+ ReceiveDouble (double *, long) : bool
+ ReceiveDoubleArray (double *, size_t, long) : bool
- SocketCleanup () : void
+ UdpReceive ()
+ ~UdpReceive ()

Figure 49 - Classes Required for the Implementation of UDP

The UdpSend class is used to send single double values over UDP datagram, while the UdpReceive class is used to receive single double values or double arrays (of customizable size) from UDP datagram. The Udp class is a convenience class used by both UdpSend and UdpReceive for the initialization and status checking of Winsock. Since the relevant simulation data is only obtained from FSX when it has changed in the flight simulation, the accelerations and velocities are not considered to be outputted from FSX as arrays, but rather as single double values. Because of this, it is convenient to be able to send simulation data one double at a time using the *SendDouble* method in the UdpSend class. Once the data has been processed by the

classical WF algorithm inside Simulink, the lengths, velocities and accelerations of the motion platform actuators are returned. The lengths, velocities and accelerations are broadcasted from the Simulink environment as arrays over UDP using Simulink's UDP Send block. As such, it is important to be able to receive double arrays that contain these values. The *ReceiveDouble* method in UdpReceive is used to receive those double arrays. In Figure 50, we can see the position of the UdpSend and UdpReceive classes in the C++ implementation. The sequence of a typical UDP communication between the C++ implementation and Simulink can also be seen.

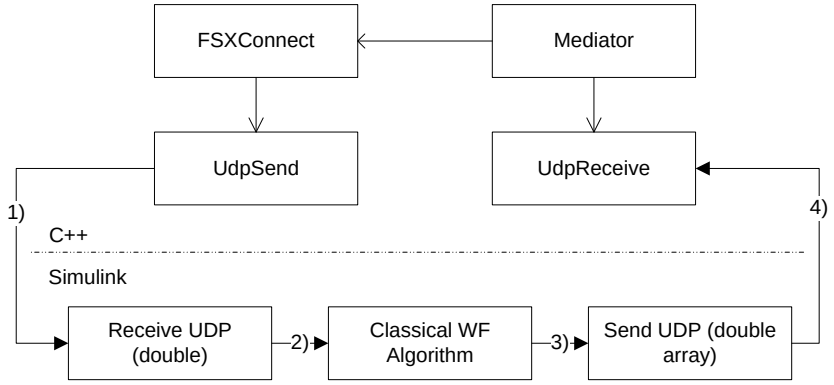


Figure 50 - Typical UDP Communication

The UdpSend class is used by FSXConnect to send the double values containing the translation acceleration and angular rates as they are received from FSX. The UdpReceive class is used by the Mediator to receive the double arrays containing the motion platform actuator lengths, velocities and accelerations from the Simulink environment. The UDP communications inside Simulink are easily implemented using the UDP receive and send blocks from the Instrument Control Toolbox. All UDP communications are conducted on the local-host (IP 127.0.0.1), which means that they are conducted on the same machine (loopback network connection). The order of data used for all UDP communications is the "Little-endian" standard. This simply means that the least significant byte of the data is always stored at the lowest address

in memory. In order to simplify the UDP communications, it was decided to use a distinct port for each variable in the system. The ports used for the transfer of all data over UDP can be seen in Table 8.

Table 8 - Port Allocation for UDP Traffic

Data	Data Type	Port
X axis acceleration	Double	10101
Y axis acceleration	Double	10102
Z axis acceleration	Double	10103
P Angular velocity	Double	10104
Q Angular velocity	Double	10105
R Angular velocity	Double	10106
Actuator lengths	Double Array	10107
Actuator velocities	Double Array	10108
Actuator accelerations	Double Array	10109
α plane pitch angle	Double	10110
β plant bank angle	Double	10111

In total, six different double values are sent from FSXConnect to the Simulink environment. Then, there are three double arrays of size three (the number of actuators for RMC's 3DOF) that are sent from Simulink to the Mediator. More information about the UDP implementation can be found in Annex K where the C++ code is available.

4.2.7 Documentation

The proper documentation of code in software implementations is essential to ensure the portability, ease of use and maintainability of any system. In addition, proper documentation can contribute to cost-effectiveness as no time is wasted when a newcomer to the project tries to

understand the system. As such, proper project documentation containing class, fields, constructors and methods information was created. This documentation allows anyone to quickly find the role of classes, methods and fields (variables) in the code. Most notably, the documentation explains which parameters are necessary for each method and explains what is returned. Although it is possible to create documentation in many formats, the Javadoc format from Sun Microsystems was selected for this study because of its simplicity and ease of use. Indeed, the Javadoc format is built upon the HTML standard and is widely used in the industry even to document code in languages other than Java. In addition to documenting the code with a Javadoc, each line of the C++ implementation was commented to ensure that a complete stranger would be able to understand any and all lines of code that were produced.

```
#include "Model.h"

/**
**  Used to establish a connection with either RMC's 3DOF or
**  6 DOF platform. Some parameters are also initialized depending
**  on which platform the connection is established with.
**
**  @param  ip  the ip adress of the controller to which a connection is attempted
**  @return     a string containing either "3DOF" or "6DOF"
*/

string Model::connect(string ip){
    ipAddress = ip;
    platformType = "3DOF";                  // Default platform type (3DOF)
        try{                                // Attempts to connect to the 3DOF
            g = new Galil(ipAddress);       // Creates a Galil Object
            g->timeout_ms = 1000;           // Allow for a 1 second timeout
            g->connection();                // Connect to the controller
            }// try
```

Figure 51 - C++ Code Sample

As can be seen in Figure 51, which shows a code sample, the highest commenting standard has been applied to the entire C++ implementation. Finally, a UML class diagram of the system complements the code documentation and comments. The UML class diagram of the system can be found in Annex D.

4.3 Software Validation

Following the software implementation phase, software validation activities were conducted to ensure that the software implementation allows the operator to conduct all of the selected use cases scenarios flawlessly. In this case, the operator needed to be able to manage a connection to the motion platform, manually control the motion platform and perform flight simulation. Two distinct software validation activities were conducted during this phase: technical verification and user validation.

4.3.1 Technical Verification

Technical verification consists of verifying each of the intended use cases thoroughly to ensure that any combination of actions performed by the users are supported by the software and that there are no errors in the operation of the system. Several tests were conducted with a single purpose: to crash or uncover errors in the system. The author of this study was able to crash the initial system several times and uncover a myriad of errors both in the code and the logic. Therefore, the technical verification phase has proven to be very effective in improving the software before its release.

First, the connection management functionalities were thoroughly tested and some errors were found under special circumstances when the motion platform controller was energized, but not the motion platform. These issues were resolved by introducing fail safes in the code. Then, the manual controls were tested to ensure that they were fully functional and that the motion platform was responding as intended. During this test it was found that some commands were inverted and that under specific conditions the software would crash. Indeed, when trying to press several manual motion command buttons in a rapid succession, the software would crash.

This discovery led to several improvements in the manual command handling code as well as the introduction of a safety feature that prevents two manual commands from being executed at the same time. Finally, the flight simulation component was thoroughly tested in combination with the manual control features to ensure that the software would comply with all user requests. Only minor issues were found during this phase and were easily corrected in the code.

4.3.2 User Validation

After having thoroughly tested each of the software's functionalities, a group of three volunteers were asked to test the software in an attempt to discover undiscovered bugs or problems. This additional testing phase, the user validation phase, is necessary to ensure that no bugs were overlooked by the author. Testing is a destructive process where one tries to find errors inside his software. As a result, it can sometimes be difficult to be truly critical of one's own work and find all the errors [24]. For this reason, submitting the software to others for testing can be an effective way of uncovering the most hidden and insidious bugs of software [24]. In this validation exercise, three volunteers in the field of Computer Science were given the task of testing all three intended use cases of the software with a mandate to attempt to crash or find bugs in the process. The goal was to unearth potential problems and to collect commentaries on the look and feel of the interface. The results of this validation exercise are summarized in Table 9.

This short and simple exercise proved to be most valuable for this study as it unearthed important problems in the software. Despite the author's best effort at debugging, some problems were still unearthed by third party reviewers. This shows the effectiveness of user validation when developing software. Finally, this exercise also provided valuable information on the expectations of the software user, a notion that is not always obvious to the software developer.

Table 9 - Observations and Problems Discovered during User Validation

Observations and Problems Discovered	Action Taken
It would be easier for the user if the controller IP Address would be detected automatically.	The standard controller IP Address is automatically typed in the appropriate field for the user.
The program crashes when commanding any motion during a parking, extend of center manoeuvre.	This problem is caused when two types of commands (absolute and jog) are executed simultaneously. Protective code is added to prevent the user from doing this.
The program connects to RMC 6DOF when there is no controller connected.	A logic error in a Try-Catch routine used to detect to which motion platform the software is connected was corrected.
The program does not detect a lost of connection from either Simulink or FSX.	A connection monitoring algorithm is implemented to ensure that motion simulation can only occur when all required connections are established and alive.
There is not enough explanation for the "Erase Parked Position" functionality. A user could damage the platform not knowing about this functionality.	The GUI was updated with further explanation to prevent inadvertent/incorrect use of this advanced feature.
If the user forgets to park the platform after a session, manual reconfiguration is required. Most users won't know how to reconfigure the platform for safe usage.	If the user forgets to park the motion platform, it will automatically park itself when closing the software.

4.4 Hardware Implementation

Following the acquisition of all the required components selected in the design phase, the hardware implementation began. This phase required the fabrication of special parts to secure the

components to the platform. Some of these parts were manufactured locally while others were machined and fabricated off-site by a commercial machining shop. This section will provide information on the origin and installation of special parts required to install the simulator components in accordance with the design. Details of the installation of the flight simulator components will also be discussed.

4.4.1 Special Mounting Parts and Installation

Special mounting parts were required to safely mount the key components selected for RMC's flight simulator onto the motion platform. Most parts requirements were driven by the concept drawings developed during the design phase as detailed in Chapter 3. Table 10 lists the special mounting parts that had to be fabricated as well as their source.

Table 10 - Special Mounting Parts

Mounting Part	Source
Seat Mount	RMC's Electrical Engineering Machining Shop
Adjustable Joystick Mount	Local Machining Shop
Adjustable Throttle Mount	Author
Adjustable Infra-red Sensor Mount	Local Machining Shop
Seat Harness Installation	RMC's Electrical Engineering Machining Shop

An aluminum seat mount was designed and fabricated by the local Electrical Engineering machining staff to safely anchor the passenger seat to the motion platform base. This mount is static and positions the seat at a height of 20 cm above the platform's surface. In addition, a

racing harness was bolted through the motion platform frame to secure the passenger to the seat. Next, an adjustable mounting system was designed and fabricated to allow for the installation of the pilot controls. The adjustable control mounts allow for the positioning of the joystick and throttle depending on the pilot's morphology and preferences. Finally, an adjustable sensor mount was fabricated to position the TrackIR 5 sensor at the appropriate height depending on the position of the pilot's head. Both the TrackIR 5 and joystick mounts had to be fabricated at a local machining shop due to their increased complexity.

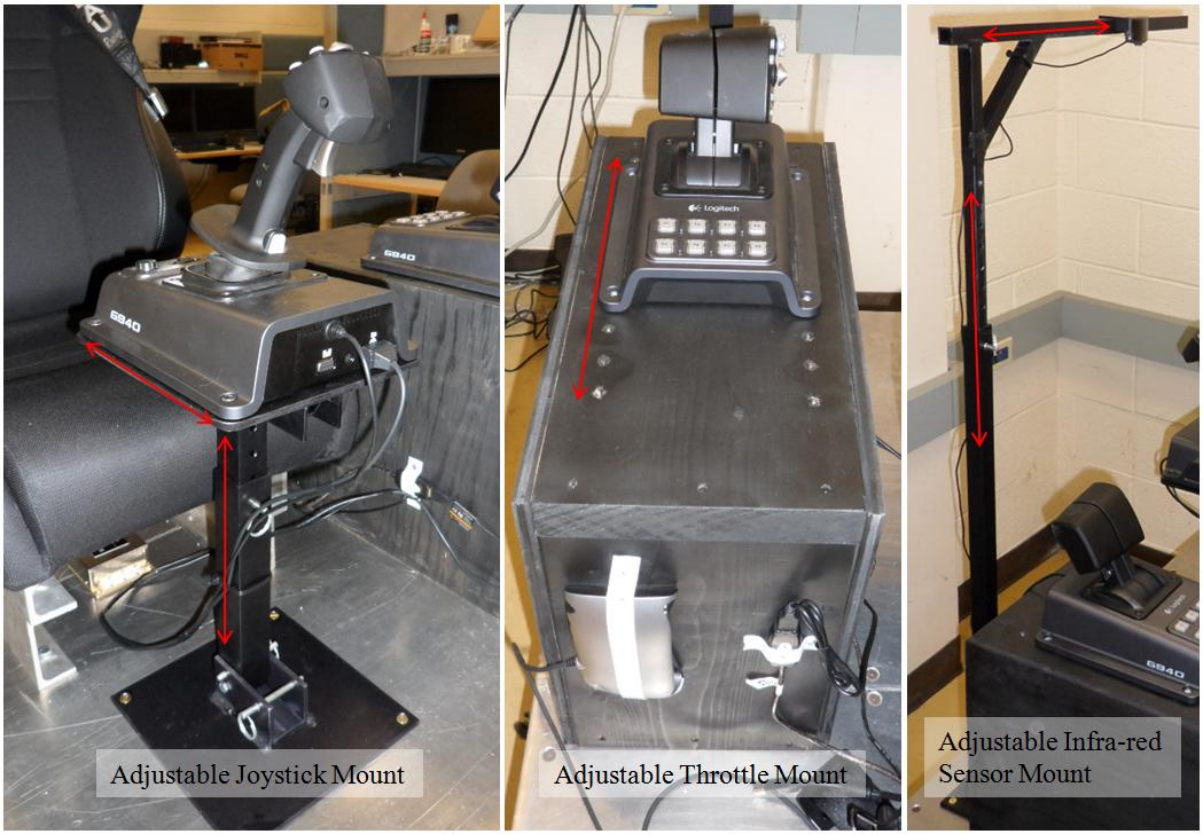


Figure 52 - Installed Special Mounting Parts

All mounting components were made from aluminum with the exception of the throttle mount block, which was fabricated out of wood by the author of this study. Figure 52 shows the

different mounting parts on RMC's 3DOF motion platform. Red arrows illustrate the directions in which the flight simulator components can be adjusted using the mounting parts.

Figure 53 - Head Mounted Display and TrackIR 5

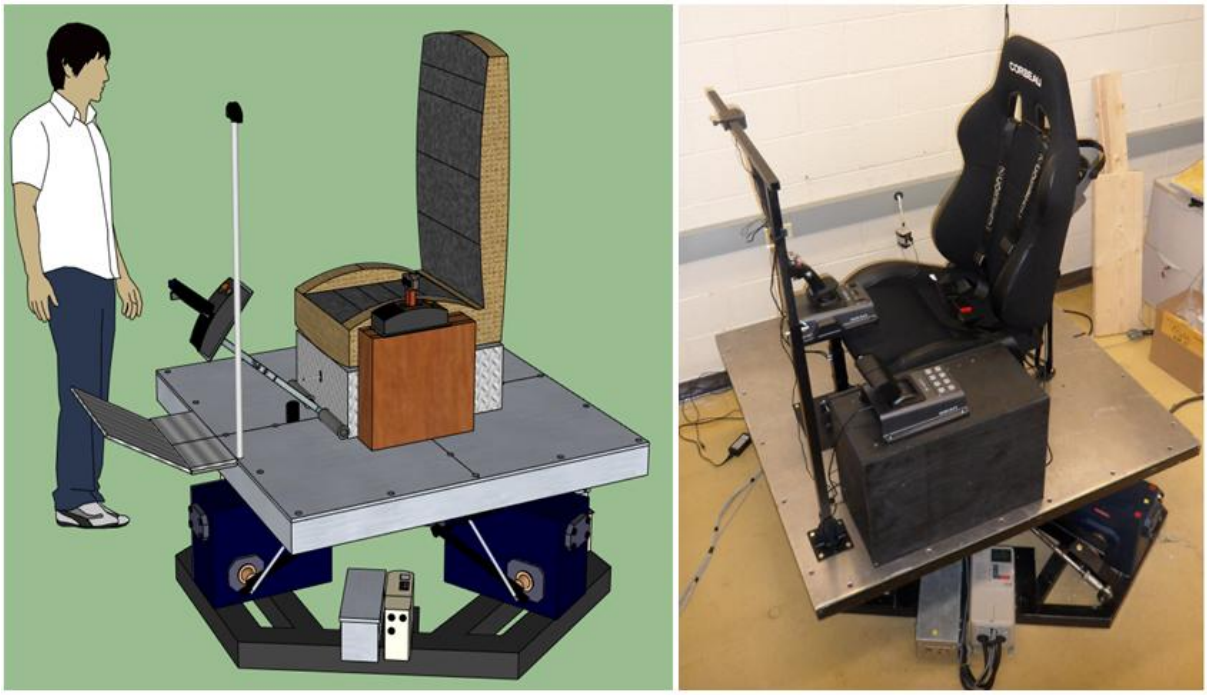


Figure 54 - Comparison between Concept Draw and Actual Implementation

Figure 53 shows the head mounted display with attached TrackIR 5 and infrared sensor. The system's key components were fastened to their respective mounting parts and cables were managed to prevent any snag during the operation. Initial testing of the flight simulator throughout its motion envelope at various speeds was also conducted to validate the installation. As can be seen in Figure 54, the final result after the installation of most components remained very close to the concept drawings that were envisioned during the design phase.

4.5 Summary

This chapter looked at the implementation phase of this study. More precisely, it covered the implementation of the software by first explaining the goals and approach of the implementation phase. Then the mediator design pattern, an essential part of the software implementation was discussed. In addition, this chapter provided the reader with an understanding of the interactions between software components, including the networking elements required to bring the system together. The leveraging of the powerful Simulink environment and its role in the software implementation was also discussed. Important references to more specific implementation material in the annexes were given to allow for a more exhaustive study of the software implementation. Finally, the implementation of the required hardware, from the process of its fabrication to its installation onto RMC’s 3DOF motion platform was detailed. The final result was also shown to be very close to the system envisioned during the design phase.

CHAPTER 5 - EXPERIMENTAL STUDY AND PILOT VALIDATION

5.1 Introduction

After having designed and implemented the desired flight simulator system, an important experimental phase began to fine tune the classical WF algorithm. This would help to study the performance of the flight simulator with the intent of improving the fidelity of motion simulation. The purpose of this chapter is to describe the experiment that was conducted to study and improve the performance of RMC's 3DOF motion platform used as a flight simulator. The experiment provided the author with a tangible way to assess the fidelity and limitations of the motion sensation generated by RMC's 3DOF motion platform when using different parameters for the classical WF algorithm. After having completed the experiment and fine tuning of the motion platform with the intent of achieving its full motion simulation potential, a validation exercise was conducted with private, commercial and military pilots. This exercise provided insight into the pilot perceived fidelity of this particular flight simulation as well as feedback on the unique design of RMC's 3DOF flight simulator.

5.2 Experimental Study of the System

The flight simulator described in Chapters 3 and 4 was run through several select flight profiles that reproduce the most prevalent manoeuvres during flight. This provided insight into the fidelity of motion the simulator generated when using different parameters for the filtering and scaling blocks in the classical WF algorithm. The data collected during this experimentation

phase allowed for the fine tuning of the classical WF algorithm. In addition, this experiment was essential in order to optimize the classical WF algorithm and to allow RMC's 3DOF motion platform to achieve its full potential as a flight simulator. In the sections to follow, the experimental design for this study will be discussed.

5.2.1 Statement of the Problem and Objectives

An experiment is usually conducted in order to gather the necessary information to address a specific problem [25]. In the case of this study, the problem of interest is to determine the optimal configuration of the classical WF algorithm in order to generate the most realistic flight motion simulation using RMC's 3DOF motion platform. Several coefficients, such as scaling and filtering coefficients, can be altered to affect the response of the motion platform. As reported by [8, 9, 10 and 12], the fine tuning of the classical WF algorithm is best accomplished through an iterative process where the different coefficients of the algorithm are modified in order to generate the best motion simulation with the available hardware. As discussed in Chapter 2, the filters and scaling coefficients of each channel interact to generate the final motion signal. With this in mind, the experiment is conducted with one overall objective: to discover the best parameters for each motion channel and how these motion channels can be combined to produce the most realistic motion simulation. As the three channels are combined to produce the final lengths, velocities and accelerations of the platform actuators, the individual effects of a channel are augmented or dampened by the other channels. The flight simulator designer must determine the scaling coefficients so as to create a balanced motion simulation where each channel contributes proportionally and realistically to the final output.

This is best accomplished when the simulator designer collects the observations of pilots between each run [4, 10]. The author of this study has more than 100 flight hours in low

performance fixed wing aircrafts. As such, the motion responses can be studied between each run by the author. To meet the objective of this experiment, the author relied on the self evaluation of the motion stimuli coupled with the use of motion sensors installed on RMC's 3DOF. It is felt that the combination of subjective and objective means of measuring motion simulation fidelity will provide the author with ample data for the effective study and optimization of the system. Indeed, the use of a motion sensor provides an additional method of measuring motion fidelity instead of following the traditional method where WF algorithm tuning relies exclusively on pilot inputs [4]. Finally, to validate the work conducted during the experiment, experienced pilots also flew the simulator during the pilot validation phase and produce additional feedback on the quality of motion simulation. This is also necessary to collect an unbiased pilot feedback sample.

5.2.2 Measures of Performance and Response Variables

When conducting an experiment whose aim is the optimization of a well-understood system such as the classical WF algorithm, it is necessary to establish robust indicators of performance [25]. In this case, the performance of the classical WF algorithm is measured in two ways. First, the quality of the motion simulation is evaluated by the author who flies the simulator between each test runs. Using this method, the pilot compares his notion of expected motion stimuli to the motion stimuli experienced on the system during each flight. This is useful to identify problems that may not be easily interpreted using numerical methods. Second, the quality of motion simulation is evaluated by measuring the fidelity of the simulator motions when compared to that of the simulated aircraft in FSX. An improvement of fidelity between each test run is an indicator that the motion response is improving (and the opposite is also true). In essence, pilot evaluation is used to target problematic areas of motion simulation while numerical methods provide the tools to truly identify and solve problems.

In order to numerically measure performance during the experiment, response variables that reflect the performance of the process must be selected. The response variables are dependant variables of the process and provide an explicit measurement of the performance of the system [25]. In this study, the response variables suitable to establish the performance of the system are first taken to be the mean absolute differences between FSX's motions ($\overrightarrow{f_{AA}}$ and $\overrightarrow{\omega_{AA}}$) and the simulated motions measured on RMC's 3DOF ($\overrightarrow{f_{SS}}$ and $\overrightarrow{\omega_{SS}}$). Two standard deviations for each motion channel are then added to each value to account for variation across the entire test run within a confidence interval of 95%. Equation 5.1 and 5.2 show the calculations used to establish the response variables for accelerations and angular rates where *n* represents the number of motion data points within a simulation run.

$$\overrightarrow{RV_A} = \frac{\sum_{i=1}^{n} \left|\overrightarrow{f_{AA(i)}} - \overrightarrow{f_{SS(i)}}\right|}{n} + 2\sigma \tag{5.1}$$

$$\overrightarrow{RV_\omega} = \frac{\sum_{i=1}^{n} \left|\overrightarrow{\omega_{AA(i)}} - \overrightarrow{\omega_{SS(i)}}\right|}{n} + 2\sigma \tag{5.2}$$

As such, six response variables were calculated to evaluate the system's performance. The response variables represent the mean margins of errors for each motion channel with a confidence interval of 95% (2σ). The sample standard deviation is used to measure dispersion for each response variable in accordance with Equation 5.3 where σ is the sample standard derivation, x_i a dataset value (i.e. $x_i = \left|\overrightarrow{f_{AA(i)}} - \overrightarrow{f_{SS(i)}}\right|$) and μ the arithmetic mean of a dataset.

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \mu)^2} \tag{5.3}$$

In theory, as the accuracy of a motion channel increases, the value of its response variable decreases. Depending on the channel under study, not all response variables share the same relevance to performance. For example, when tuning the translational acceleration channel,

the angular rate response variables won't carry much information on the performance of the channel. In addition, the yaw motion channel response variable is never relevant for this study since RMC's 3DOF is not capable of generating yaw motions. Thus, the response variables of interest are selected depending on the channel under study. Matlab code was developed to automatically calculate the response variables from a test run. This response variable calculation code can be found in Annex E. Finally, the visual inspection of motions against time plot between experimental and recorded flights can provide additional insight into the quality of motions simulation.

The motions of the flight simulator are captured using an inertial measurement unit. The Crossbow IMU 320 dynamic measurement system is specifically used to measure the translational accelerations and angular rates of the motion platform. As can be seen in Figure 55, it is installed at the simulator reference frame F_S, which is located in the geometrical center of the motion platform below the pilot seat. As discussed in Chapter 2, reference frame F_S must be in the same location with respect to the pilot in both the simulated aircraft and the motion platform. As such, the measurements provided by the IMU 320 ($\overrightarrow{f_{SS}}$ and $\overrightarrow{\omega_{SS}}$) are taken at the same location as $\overrightarrow{f_{AA}}$ and $\overrightarrow{\omega_{AA}}$ (the inputs to the classical WF algorithm). In order to conduct the experiment with ease and precision, the IMU 320 was interfaced directly into the Simulink environment using a serial connection through the RS-232 connector of the unit. Specific details on the implementation of an interface to receive the IMU 320 signal into the Simulink environment are discussed in Annex F. Table 11 shows the relevant performance data of the IMU 320.

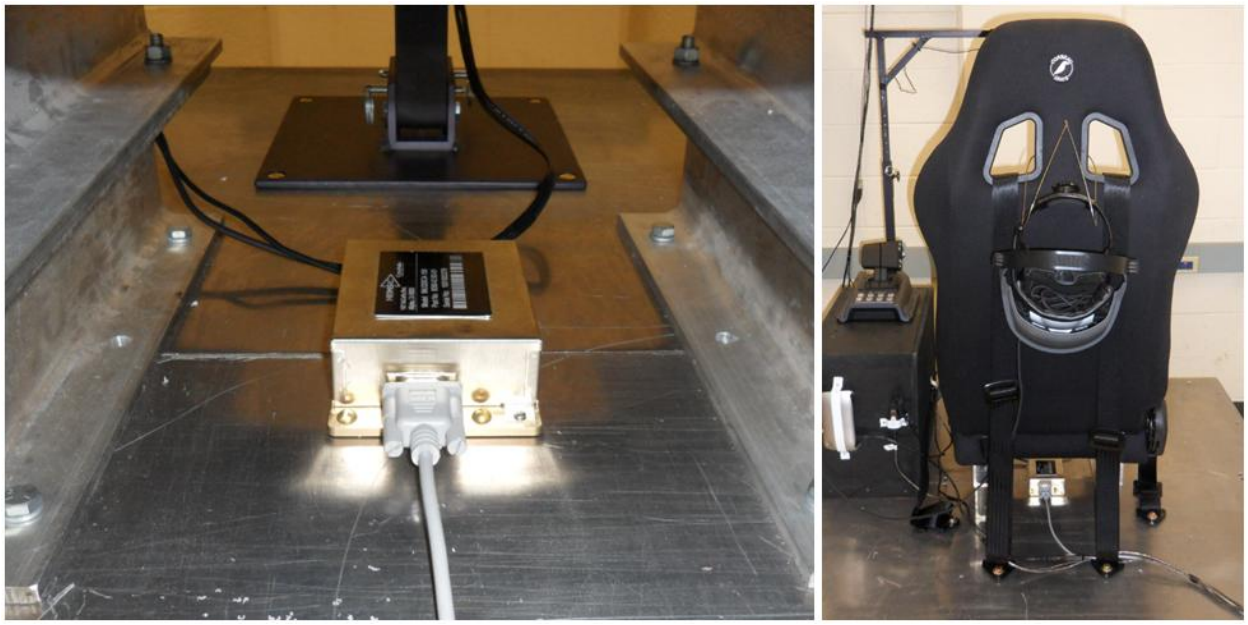


Figure 55 - The Position of the IMU 320 on RMC's 3DOF Motion Platform

Table 11 - Crossbow IMU 320 Performance Data

Performance	Acceleration	Angular Rate
Range	± 4 *g*	± 150 $deg \bullet s^{-1}$
Resolution	< 0.5 *mg*	< 0.02 $deg \bullet s^{-1}$
Bandwidth	20 Hz	20 Hz
Scale Factor Error	< 1%	< 1%

As can be seen from the performance data of the IMU 320, it is well suited for this experiment as it is capable of measuring acceleration and angular rate at ranges and resolutions well beyond the values that will be encountered during the tests.

5.2.3 Choice of Factors

Factors represent the independent variables that are varied throughout the experiment to affect its outcome. Typically, these variables are varied over a pre-selected range established by

the experimenter based on his knowledge of the process and theorical considerations [25]. In this study, the factors of interest are the parameters that define the behaviour of each motion channel. Scaling coefficients and filtering equations for each motion channel represent these parameters. In addition, the rate limiting coefficient of the tilt-coordination channel is also of interest. This coefficient limits the rate at which tilt-coordination can occur. Typically, the scaling factors of a motion channel will range non-inclusively from 0 to 1. Tilt-coordination rate limiting is typically between *2.6* and *3.6* $deg \bullet s^{-1}$ [3] to remain below the threshold of human sensory detection for rotational velocity. It is harder to define a range for filtering equation coefficients since their transfer functions have many attributes such as form, degree, natural break frequency and damping ratio. A complete study on the effect of each filter coefficient for the tuning of the classical WF algorithm can be found in [12]. Based upon the findings of [12], it would be unproductive to create an experiment where a list of these filtering variables were selected and methodically tested in order to discover the best results. As such, this study is conducted with a sequential approach where observations are made between each test runs and where the parameters deemed appropriate for the next iterations are formulated from the results of the previous iterations. In fact, as reported by [12], an iterative process of filter tuning is the most appropriate form of optimization for the classical WF algorithm. By observing the behaviour of a specific channel after each test run, the filtering equations can be modified in a deliberate fashion to improve the quality of motions generated by the platform.

5.2.4 Experimental Design

In this study, the classical WF algorithm is studied and optimized as a whole by playing back a series of pre-recorded flight profiles through the system. Then, by comparing recorded

motions to those of the platform, the fidelity of motion simulation can be studied and optimized in order to take full advantage of RMC's 3DOF motion platform capabilities.

The Cessna 172 SP Skyhawk (C-172) is selected as the aircraft for which we will attempt to simulate motion. The C-172 is selected because it is a small and very popular aircraft in the civilian aviation world. In addition, it is widely used as a pilot trainer in most civilian flight schools [27] and some military establishments [28]. For this reason, it is interesting to study the degree of realism that can be achieved using this airplane and RMC's 3DOF motion platform.

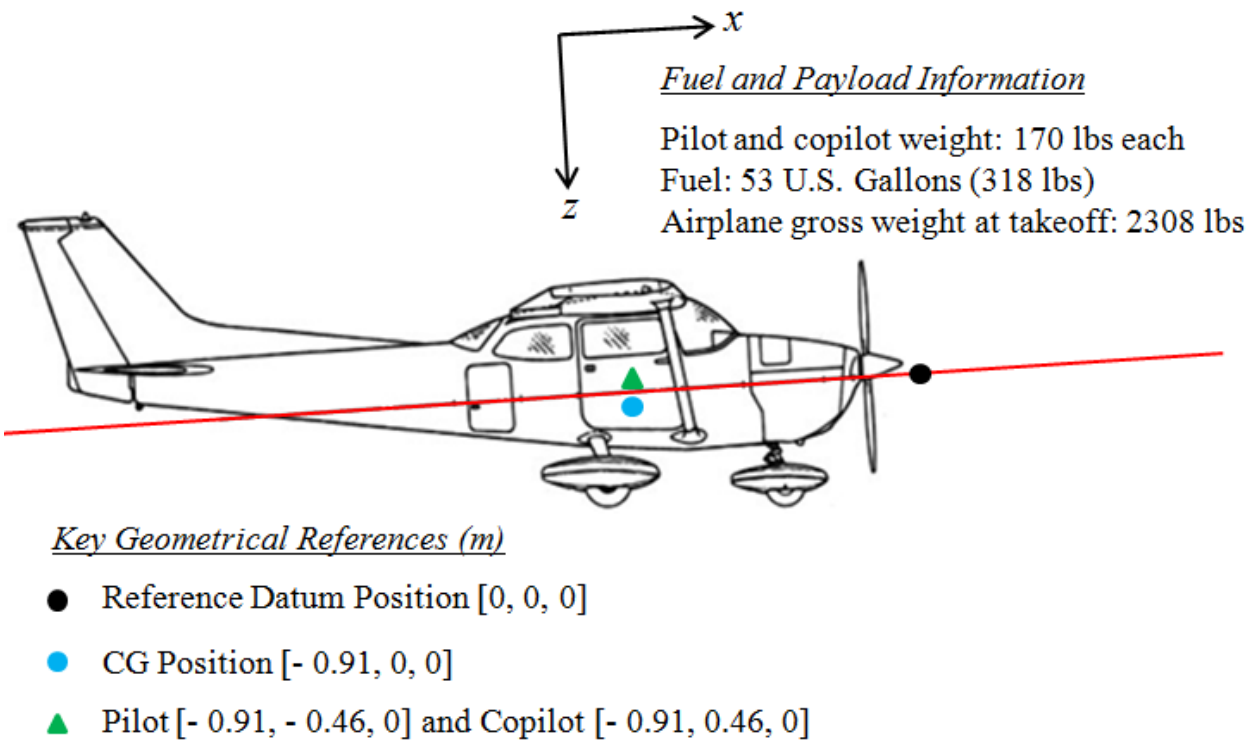

x
z
Fuel and Payload Information
Pilot and copilot weight: 170 lbs each
Fuel: 53 U.S. Gallons (318 lbs)
Airplane gross weight at takeoff: 2308 lbs
Key Geometrical References (m)
Reference Datum Position [0, 0, 0]
CG Position [- 0.91, 0, 0]
Pilot [- 0.91, - 0.46, 0] and Copilot [- 0.91, 0.46, 0]

Figure 56 - FSX's C-172 - Key Information and Geometrical References

Positive results would mean that the system under study could be of use for the simulation of such airplanes. Finally, because of its popularity, it is also very easy to find qualified pilots for a validation exercise of the flight simulator. Figure 56 shows key information and geometrical references for the C-172 model used in this study. It is important to note from

Figure 56 that the position of the pilot is slightly to the left of the aircraft's center of gravity (CG). The accelerations and angular rates obtained from FSX are from the CG position. In order to compare the motions of the FSX aircraft with those of RMC's 3DOF motion platform, both motion measurements must be taken at the same position with respect to the pilot. In this case the motion measurements are taken at reference frame F_A for the simulated aircraft and reference frame F_S for the flight simulator. The motions recorded for the simulated aircraft in FSX are however reported from the center of gravity of the aircraft which is represented by reference frame F_B. Therefore, for the C-172 it is necessary to use classical mechanics to determine the accelerations that are generated slightly left (46 cm) of the CG position where reference frame F_A can be found under the pilot's seat. It is well known that in a rigid body, the angular rates are the same for any point of the body [29]. However, translational accelerations a distance away from the CG are affected by angular accelerations of the body around its CG. To obtain the accelerations at an arbitrary point from the CG in a rigid body it is necessary to account for translational accelerations of the body, normal accelerations and angular accelerations. Equation 5.4 [29] is used to calculate the accelerations at the pilot's position for each time step where $\overrightarrow{P_A}$ is a vector between the CG and the pilot's location, $\vec{a}_{CG}$ is the acceleration of the aircraft at the CG, $\vec{\varepsilon}_{CG}$ is the angular acceleration of the aircraft at the CG and where $\vec{\omega}_{CG}$ is the angular velocity of the aircraft at the CG. By doing this, the pilot's location with respect to the point where motions are measured is the same on the aircraft and on RMC's 3DOF (F_A and F_S). It is also the point, F_S, where the Crossbow IMU is installed on the flight simulator for this experiment.

$$\vec{a}_{Pilot} = \vec{a}_{CG} + \vec{\varepsilon}_{CG} \times \overrightarrow{P_A} + \vec{\omega}_{CG} \times (\vec{\omega}_{CG} \times \overrightarrow{P_A}) \tag{5.4}$$

Using the C-172, three flight profiles representing the most prevalent manoeuvres in flight are recorded from the FSX environment. The flight profiles are recorded from the

following manoeuvres: a continuous climb, a standard rate turn and a descent. Each flight profile is of duration of 30 seconds. The flight profiles were recorded using Simulink and manually flown by the author in the FSX environment. A short description for each of the experimental flight profiles can be found in Table 12.

Table 12 - Experimental Flight Profiles Manoeuvres

Manoeuvre	Description
Climb	Entry into a continuous 1000 feet/min climb from level flight.
Standard Rate Turn	From level flight into a 20° bank turn and back into level flight.
Descent	From level flight into a 1000 feet/min descent and back into level flight.

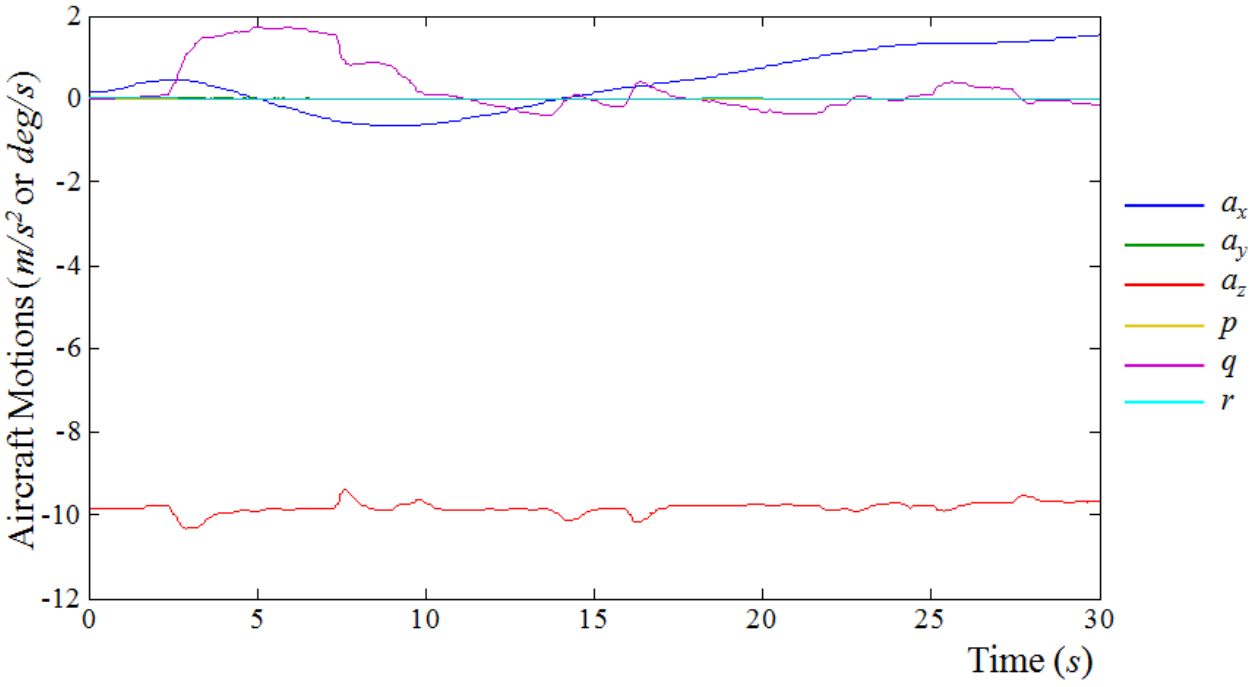


Figure 57 – Plot of Climb Profile Motions against Time

The difference between motions signals for each flight profiles can be seen in Figure 57, Figure 58 and Figure 59.

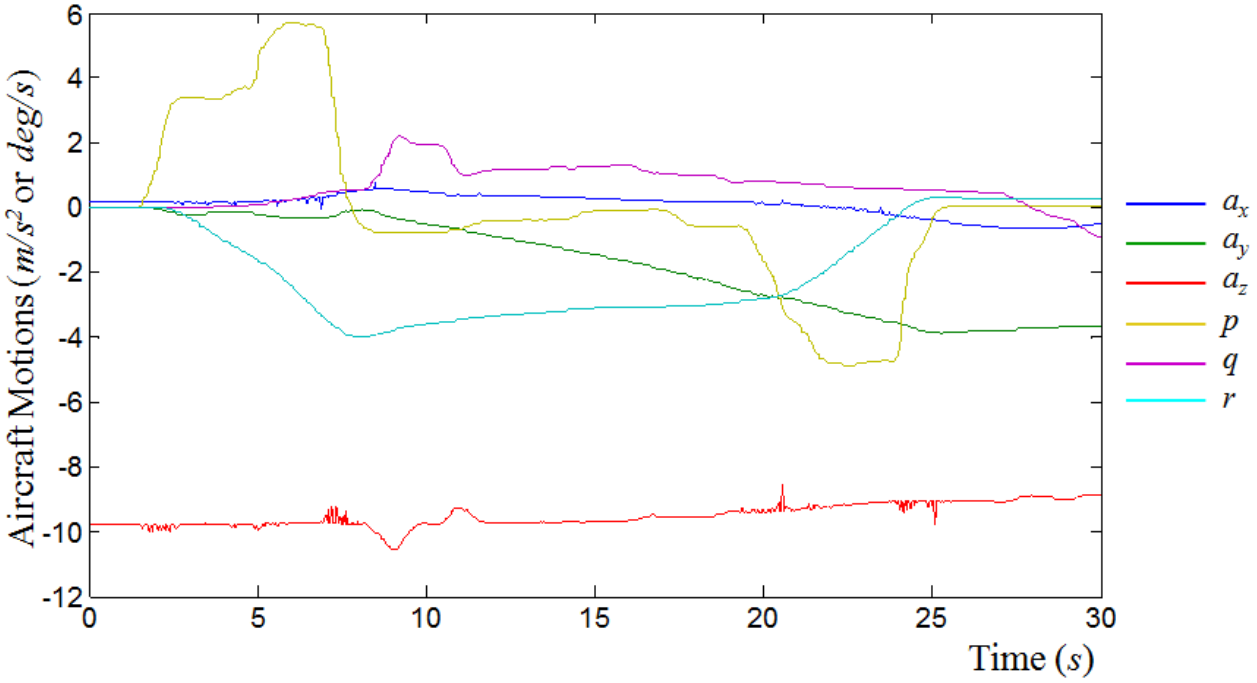


Figure 58 - Plot of Standard Rate Turn Profile Motions against Time

As a starting point, each motion channel is first configured using the optimal scaling factors and filtering equations (Table 2) as reported by [12]. For each test run, the recorded flights are "played back" through a modified version of the classical WF algorithm and the motion response of RMC's 3DOF motion platform is recorded using an inertial measurement unit. More information on the flight recording and playback modifications of the Simulink classical WF algorithm can be found in Annex G. Following a sequential approach, after each test run the performance of each channel is analyzed based on pilot feedback, motion plots and the response variables representing the mean absolute error plus two standard deviations. The motion channels parameters are then modified in order to solve problems and increase the motion

simulation fidelity. There are no hard limits on the number of tests runs required to achieve the best performance.

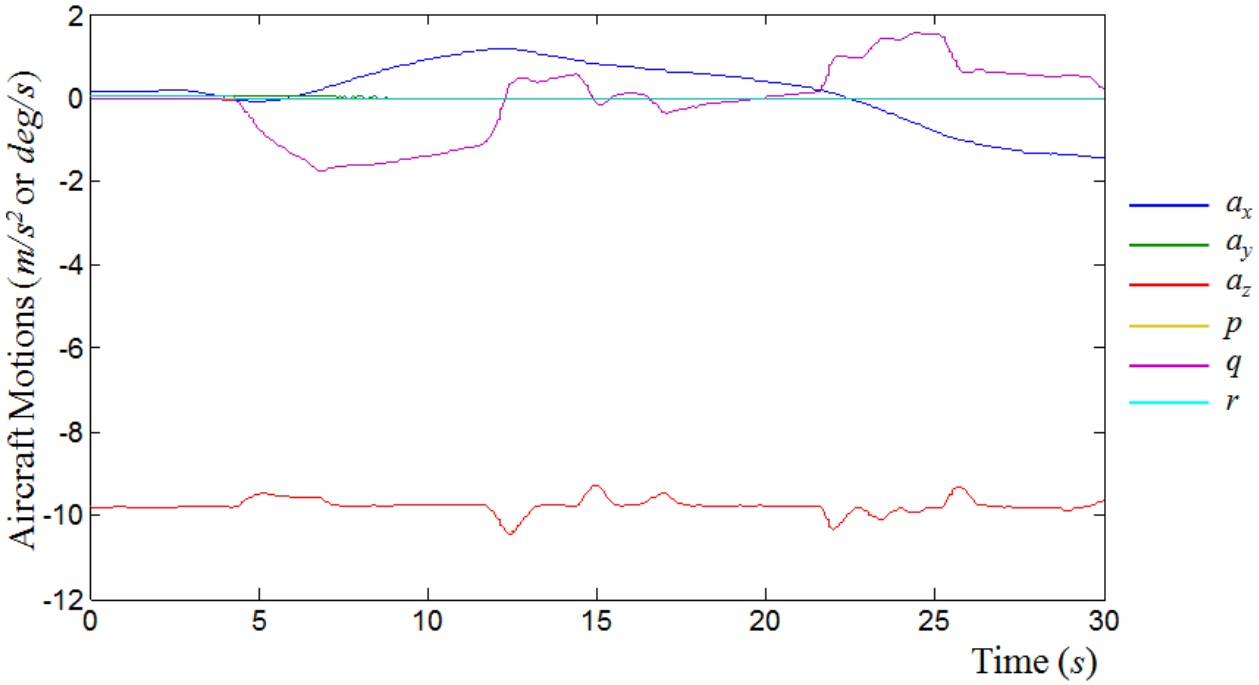


Figure 59 - Plot of Descent Profile Motions against Time

Once the experimenter has optimized the scaling and filtering blocks, and when gains on performance are deemed no longer possible, the classical WF algorithm is considered to be optimally tuned. After having studied all channels and achieved optimal tuning for RMC's 3DOF motion platform, the system is evaluated by pilots to collect additional information on performance.

5.2.4 Experimental Procedure

To protect the experimental validity of results between each test run it is important to follow a rigorous and exact protocol each time. In order to determine the best coefficients for the classical WF algorithm, the experimenter must be certain that the results observed between each run depends only on the modification applied to the system and are not the result of an

experimental error. A precise experimental procedure is also important to ensure the reproducibility of an experiment by others. Table 13 shows a step-by-step list of the experimental procedure used for each test runs.

Table 13 - Experimental Procedure for Each Test Runs

Step	Description
1	Configure the Classical WF algorithm with the scaling and filtering coefficients under study.
2	Ensure that all equipment (motion platform, inertial measurement unit) is energized and ready for the test run. Center the motion platform if required.
3	Ensure that the appropriate flight profile is set to be read in the Simulink Playback model. Set the simulation time accordingly (30 seconds)
4	Ensure that the Inertial measurement unit recording block is powered on and properly configured to save the data under the appropriate filename.
5	Begin the motion simulation using the Playback Simulink model.
6	Once the simulation is complete, save the test run results.
7	Calculate the response variables and analyze the data.

This procedure represents the actions taken during one test run for each channel and the overall system. In accordance with the sequential approach of this experiment, the data and response variables for each test run are analyzed in order to determine the scaling and filtering coefficients of interest for the next run. Unlimited flight testing by the author is also permitted to

collect a subjective measure of performance in order to better target problematic areas of motion cueing. Observations are collected in writing and kept as experimental data.

5.3 Conduct of the Experiment

After having carefully designed the experimentation used to optimize the classical WF algorithm, experimental test runs were conducted. Prior to the experimental test runs, due diligence was exercised by thoroughly testing the precision of the system. This precision was measured through a series of control flights. In this section, test runs for each motion channel are discussed in the order in which they were conducted. As predicted in the experimental design, several runs were required to reach a satisfactory level of performance. The optimization of each motion channel was accomplished and lead to a higher fidelity in motion simulation. Due to the sequential nature of this optimization process, the analysis of results was conducted immediately after each test run to derive the coefficients required to improve performance.

5.3.1 Reproducibility and Repeatability Considerations

The experiment in this study is essentially designed to measure and improve the accuracy of the flight simulator. However, before measuring accuracy, the flight simulator's precision must be measured to ensure the trueness of data collected during the experiment. In this case, precision or reproducibility represents the degree to which the flight simulator will generate the same motions for each run of an identical flight profile. Ideally, the motion simulator should always generate exactly the same motions between each run of an identical flight profile. If it is the case, the experimenter then knows with certitude that recorded variations between each run are caused solely by changes to the coefficients of the system that are modified, and not by randomness or instability in the system. In reality, small variations make their way to the system

and the motions generated by the flight simulator are slightly different between each run of an identical flight profile. With that in mind, a small experiment was conducted to measure the loss of precision that can be expected for each motion channel. Using Simulink, a flight profile was recorded and played back three times through the classical WF algorithm. For each run, the motions generated by RMC's 3DOF motion platform were recorded using the Crossbow IMU. An average motion dataset was then generated from the three recorded sets. Next, the absolute error between each of the recorded sets and the average set was calculated. These absolute error dataset for each run were averaged to create an absolute mean error dataset. Finally, the mean error for each motion channels of this mean error dataset was calculated along with the standard deviation. As can be seen in Table 14, a margin of error with a confidence interval of 95% was calculated from this data.

Table 14 - Margin of Error for Three Identical Runs (Unsmoothed Signals)

Motion Channel	Mean Error	Standard Deviation	Margin of Error (95% Confidence Interval)
Translational *x*	± 0.0227 $m \bullet s^{-2}$	*0.0315* $m \bullet s^{-2}$	± 0.0857 $m \bullet s^{-2}$
Translational *y*	± 0.0142 $m \bullet s^{-2}$	0.0148 $m \bullet s^{-2}$	± 0.0438 $m \bullet s^{-2}$
Translational *z*	± 0.0223 $m \bullet s^{-2}$	0.0256 $m \bullet s^{-2}$	± 0.0735 $m \bullet s^{-2}$
Angular *p*	± 0.2218 $deg \bullet s^{-1}$	0.1367 $deg \bullet s^{-1}$	± 0.4952 $deg \bullet s^{-1}$
Angular *q*	± 0.2430 $deg \bullet s^{-1}$	0.1598 $deg \bullet s^{-1}$	± 0.5626 $deg \bullet s^{-1}$
Angular *r*	± 0.1682 $deg \bullet s^{-1}$	0.1014 $deg \bullet s^{-1}$	± 0.3710 $deg \bullet s^{-1}$

At first glance, the loss of precision on translational channels is much lower than the lost of precision on angular rate channels. This can be explained by the fact that the Crossbow IMU used to measure the motions of the platform exhibits an especially noisy signal for angular rate measurements. As can be seen in Figure 60, where an acceleration signal in $m \bullet s^{-2}$ is compared to an angular rate signal in $deg \bullet s^{-1}$, the translational acceleration signal exhibits sporadic and low amplitude noise. On the other hand, the angular rate signal exhibits a prevalent noise of an amplitude $\cong 0.5\ deg \bullet s^{-1}$. Signal noise is reflected in the margin of error calculated in Table 14, as the slightest phase shift between the flight profile motions and the Crossbow IMU outputs will generate an error similar to the noise amplitude. A phase shift between the signals could be caused, for example, by the system's latency.

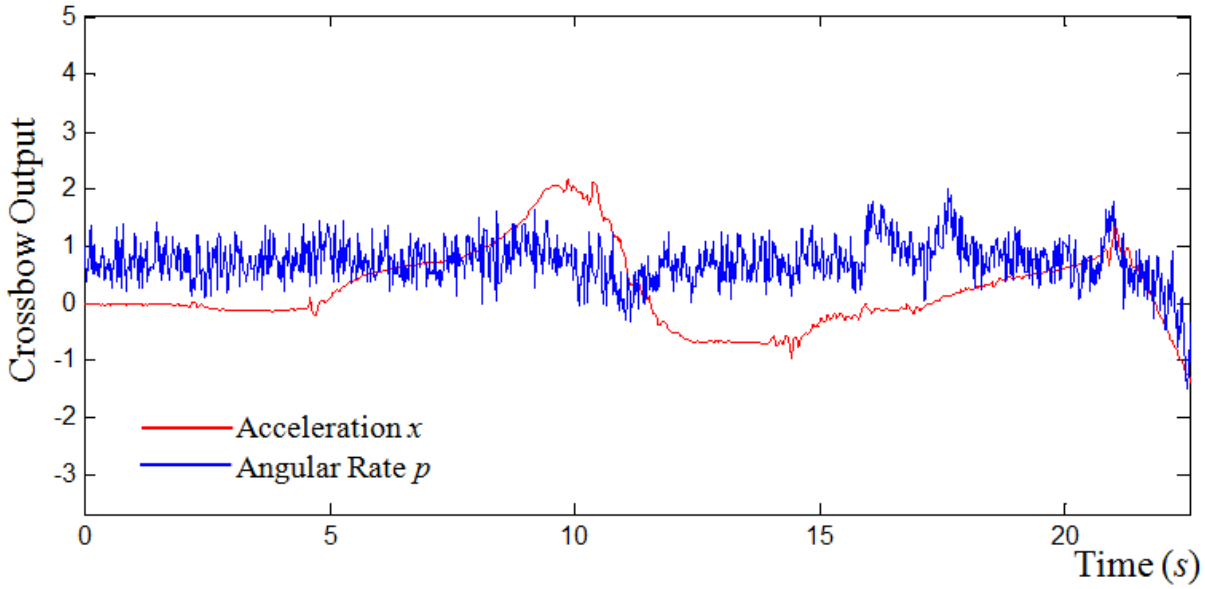


Figure 60 – Noisy Crossbow Crossbow Angular Rate Signal

Thus, the weakest link of the system is the Crossbow IMU, which exhibits noisy signals for angular rates. This lost of precision can be reduced by smoothing the Crossbow IMU signals to remove the high frequency noise. In this case, it is appropriate to do so because the expected

signal is of a smooth nature. Indeed, the noise observed is not expected when observing a flight simulation system such as the one under study and is not caused by the motion platform motions as it exists even when the IMU is perfectly still. For this reason a linear moving window average smoothing function described by applying Equation 5.4 twice on each data point of the Crossbow IMU outputs is used to smooth the signal [30]. For this study, acceptable noise attenuation without a significant loss of precision is achieved with a moving window average width of nine data points.

$$S_j = \frac{y_{j-4}+y_{j-3}+y_{j-2}+y_{j-1}+y_j+y_{j+1}+y_{j+2}+y_{j+3}+y_{j+4}}{9} \tag{5.4}$$

Figure 61 shows a typical angular rate Crossbow IMU signal and the resulting smoothed signal. As can be seen, this smoothing technique is very effective at neutralizing the noise present in the signals. However, due to the moving average nature of this smoothing technique, it cannot be applied to the extremities of a signal because four values are required before and after a data point to apply the smoothing Equation. As such, the first and last four data points of a signal are always zeroed by this technique. To work around this problem, it is convenient to simply average the unsmoothed data points and replace each of them with the mean value obtained. In this case, such extremity errors are acceptable because only 0.08 seconds of data at each extremity of a signal remains unsmoothed (the system's frequency is 50 Hz). The signals initially collected from three identical runs were smoothed to evaluate the effectiveness of this technique to improve performance. As can be seen in Table 15, the smoothing of Crossbow IMU signals has a positive effect on the overall precision of the system. Even if the angular rate signals were the most affected by noise, smoothing was applied to all Crossbow IMU signals. As a result, a gain on precision was also made on the translational motion outputs of the Crossbow IMU.

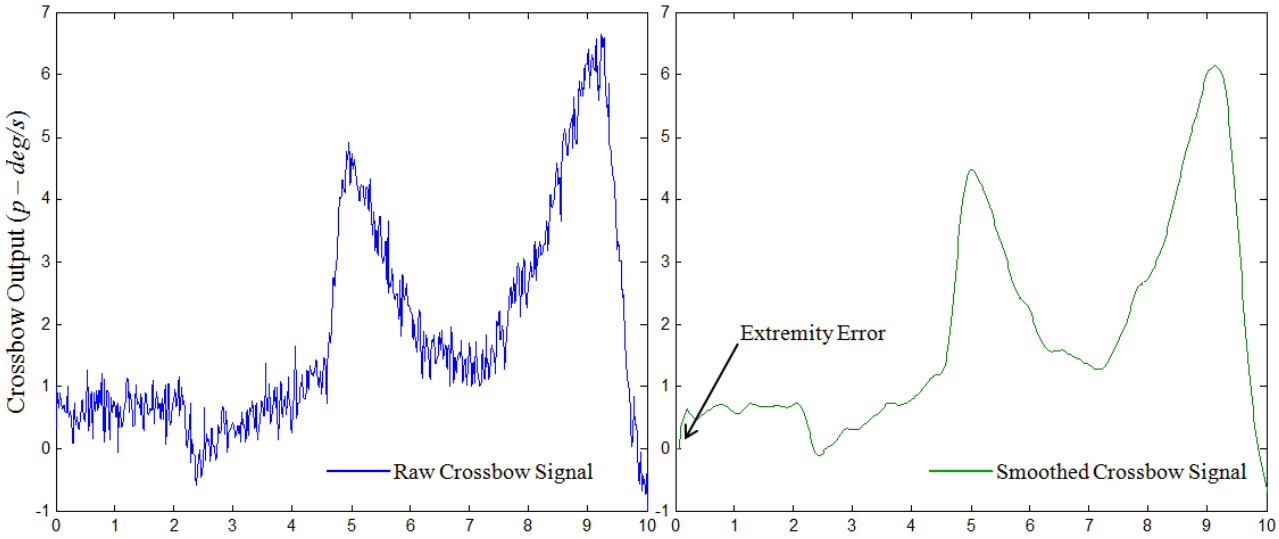


Figure 61 - Smoothing of a Typical Crossbow Angular Rate Signal

This gain on precision can be easily visualized in Figure 62, where margins of errors of raw and smoothed signals are compared. Indeed, using this smoothing technique, precision is at least doubled on each channel.

Table 15 - Margin of Error for Three Identical Runs (Smoothed Signals)

Motion Channel	Mean Error	Standard Deviation	Margin of Error (95% Confidence Interval)
Translational *x*	± 0.0096 $m \cdot s^{-2}$	0.0144 $m \cdot s^{-2}$	± 0.0384 $m \cdot s^{-2}$
Translational *y*	± 0.0043 $m \cdot s^{-2}$	0.0073 $m \cdot s^{-2}$	± 0.0188 $m \cdot s^{-2}$
Translational *z*	± 0.0041 $m \cdot s^{-2}$	0.0097 $m \cdot s^{-2}$	± 0.0236 $m \cdot s^{-2}$
Angular *p*	± 0.0754 $deg \cdot s^{-1}$	0.0608 $deg \cdot s^{-1}$	± 0.1971 $deg \cdot s^{-1}$
Angular *q*	± 0.1040 $deg \cdot s^{-1}$	0.0996 $deg \cdot s^{-1}$	± 0.3031 $deg \cdot s^{-1}$
Angular *r*	± 0.0451 $deg \cdot s^{-1}$	0.0249 $deg \cdot s^{-1}$	± 0.0948 $deg \cdot s^{-1}$

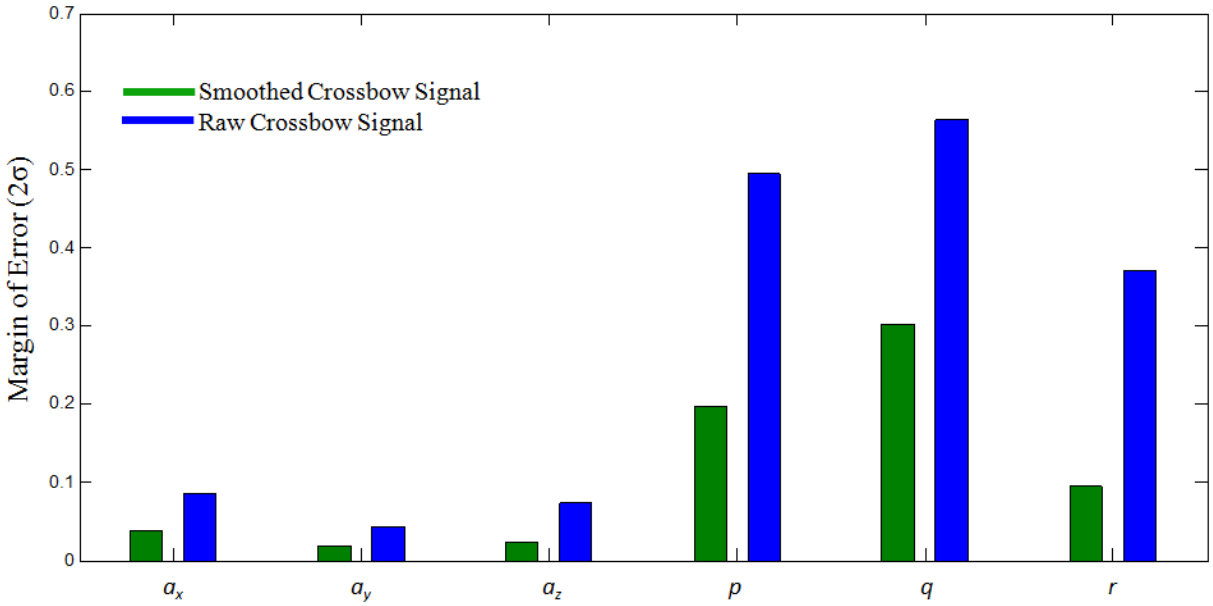


Figure 62 - Margin of Errors (Smoothed vs. Raw Crossbow Signals)

As such, it was decided that all Crossbow IMU channels would be smoothed during the experiment. Having studied the precision of the system and the measurement technique used during the experiment, it is deemed that sufficient precision is exhibited to proceed with the experiment. This precision guarantees a high degree of repeatability between each test run. In light of these facts, it is decided that a recording for each test run is sufficient to establish a significant measure of performance. Performing several recordings and averaging over each test run would increase the time required to conduct the experiment and would not provide any substantial gain in the quality of the data.

5.3.2 Test Run #1

The first test run featured the configuration with the most favourable pilot responses from [4 and 12]. This configuration can be seen in Table 16. In essence, this test run was an attempt to reproduce the conditions that received good pilot feedback for the Classical WF algorithm [4], and to evaluate how well they work with RMC's 3DOF.

Table 16 - Classical WF algorithm Configuration (Test Run #1)

Parameters	Scaling/Limiting	Filter Parameters
a_x	0.5	$\omega_n = 5, \zeta = 1$
a_y	0.5	$\omega_n = 8, \zeta = 1$
a_z	0.5	$\omega_n = 4, \zeta = 1$
p	0.5	$\omega_n = 1$
q	0.5	$\omega_n = 1$
r	0.5	$\omega_n = 1$
Tilt-Coordination	$3\ deg \bullet s^{-1}$	-

Although it is not expected that this configuration will be best suited for RMC's 3DOF, it serves as a starting point from which more appropriate coefficients will eventually be derived. All three flight profiles were played back as per the experimental procedure.

To begin this test run, the author sat in the simulator and flew several flight manoeuvres to collect subjective information on the performance of the configuration selected. Overall, the motion simulation for this test run was felt to be clearly inadequate and even detrimental to the overall flight simulation. Pilot observations for this test run are summarized in Table 17. Table 18 shows the response variables obtained for each run. It is important to note that not all response variables provide relevant information for each flight profile. Indeed, depending on the manoeuvre, some motion channels are not significantly stimulated. As such, a color code is used to identify response variables with a high relevance to the study of each motion channel.

Table 17 - Pilot Observations (Test Run #1)

Motion Channel	Observations
Translational	Large heave overshoot are present after a heave motion. (i.e. After a heave up, the platform will travel down forcefully without any reason.)
Tilt-Coordination	Tilt-Coordination is nearly non-existent. Only small angles can be observed even on aggressive cabin tilts.
Rotational	Rotational motion appears to be proportional. However, both pitch and roll motions appear underpowered.

Table 18 - Response Variables for Each Motion Channels (Test Run #1)

Flight Profile	Response Variables for Each Motion Channels Relevance: Translational, Tilt-Coordination, Angular					
	a_x $(m \bullet s^{-2})$	a_y $(m \bullet s^{-2})$	a_z $(m \bullet s^{-2})$	p $(deg \bullet s^{-1})$	q $(deg \bullet s^{-1})$	r $(deg \bullet s^{-1})$
Continuous Climb	**1.3336**	0.0331	**0.2829**	0.2437	**1.5822**	0.1601
Standard Rate Turn	0.4830	**4.0451**	0.9405	**4.4526**	2.1189	4.8868
Descent	**1.5382**	0.0272	**0.3582**	0.2744	**1.8593**	0.1625

For this test run, the response variables (Equations 5.1 and 5.2) serve a limited purpose by establishing a baseline level for comparison. They will be of great value after the second test run as they will serve to objectively compare performance between test runs. In future test runs, motion channel performance improvements should be characterized by lower response variables.

In light of pilot observations, experimental plots are studied to determine the possible causes and solutions of motion cueing problems. The descent flight profile, featuring the largest translational acceleration error of *0.3582* $m \bullet s^{-2}$, is ideal to investigate heave overshoot problems. As can be seen in Figure 63, the heave overshoot problem found during pilot evaluation can also be found in the plot of motions. An example of such an occurrence is identified on the plot of motions with a blue arrow. In the example selected, it can be seen that the overshoot also has a cumulative effect as it lowers the starting point of the next acceleration peak.

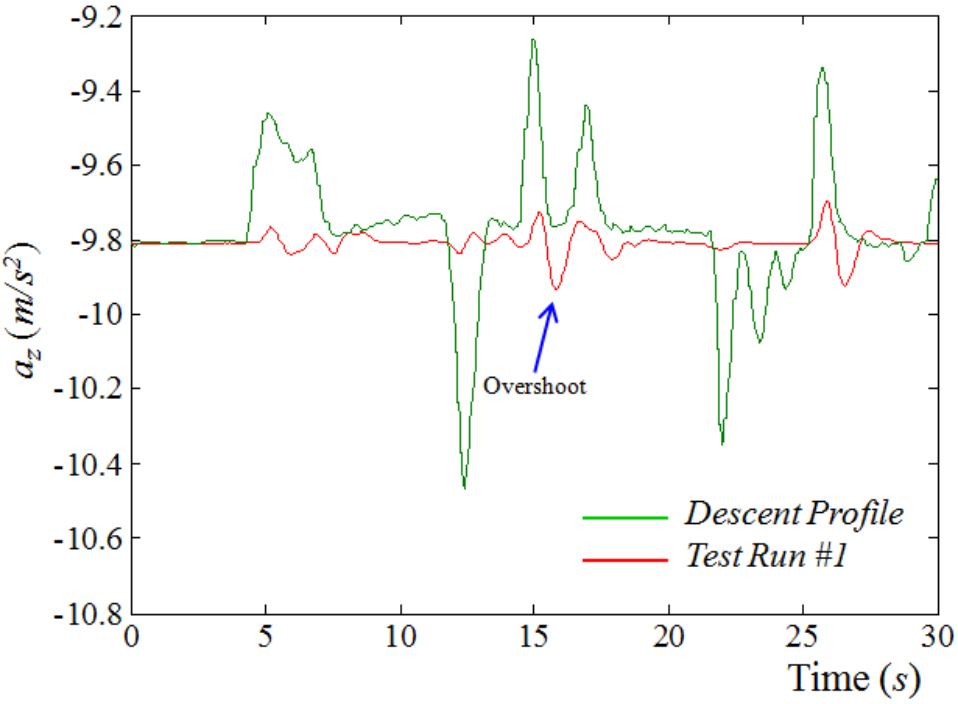


Figure 63 – *z* Axis Accelerations (Climb Profile vs Test Run #1)

According to a study conducted by [12], this problem is inherent to the use of second order high pass filters. It is possible to slightly reduce overshoot effects by increasing the filter damping ratio of the Equation (ζ). In addition, the effects of overshoots can be dampened by increasing scaling on the affected channel. However, it was found during testing that these solutions reported by [12] only provide a minimal improvement of overshooting. As such, the

problem of filter overshoot was mitigated through a more radical method. As the process used in this study for neutralizing overshooting is very specific, Section 5.3.3 is dedicated to this subject.

Next, the absence of tilt-coordination (designed as a cue substitution for translational accelerations) on *x* and *y* axis translational motion channel is investigated. The profile of choice to investigate this error is the standard rate turn profile as it features the highest tilt-coordination error of *4.0451* $m \bullet s^{-2}$. As can be seen from Figure 64, the flight simulator exhibits a very weak tilt-coordination.

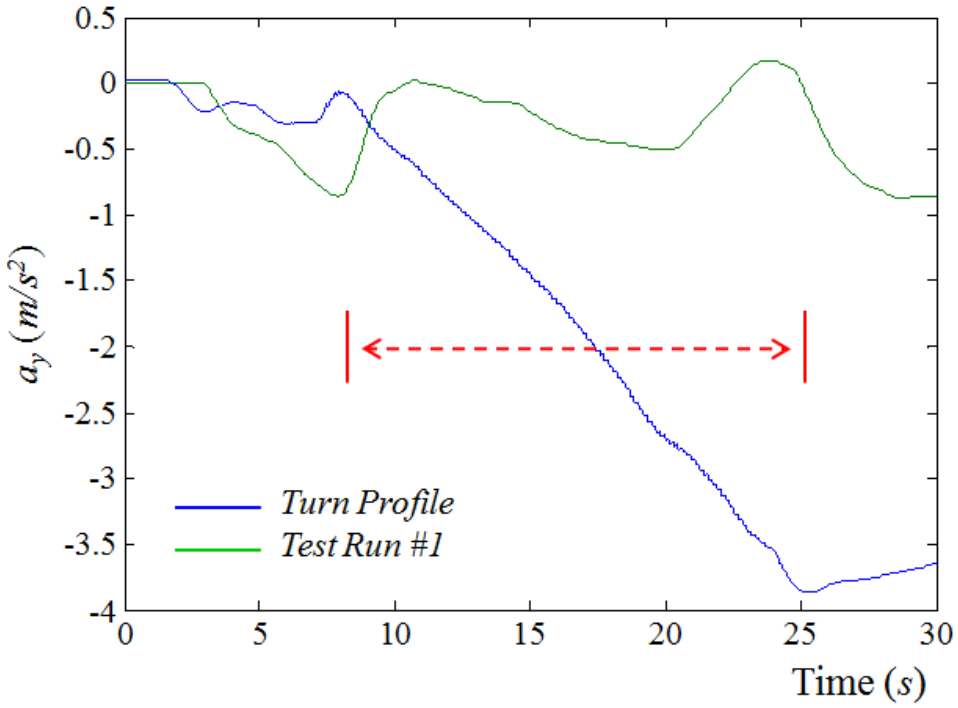


Figure 64 - *y* Axis Accelerations (Turn Profile vs Test Run #1)

The time interval in which tilt-coordination should occur is identified by red markers on the plot of motions. The two *y* axis acceleration peaks on the turn signal are caused by motion generated by the rotational motion channel as the aircraft enters and leaves the turn. After further investigation into the tilt-coordination channel, it was determined that filtering equations exhibit the appropriate behaviour. In order to allow for greater tilt-coordination, there are then two

possible options. First, scaling of the *x* and *y* axis acceleration channel can be reduced to allow for a greater amplitude in these signals. Second, the rate limiting coefficients of the tilt-coordination channel (currently at $3\ deg \bullet s^{-1}$) can be increased. Thus, the next test run features a combined increase of scaling and tilt-limiting coefficients to promote tilt-coordination.

Finally, the underpowered rotational motion channel cues are investigated. The standard rate turn profile is selected in light of its high angular rate error of $4.4526\ deg \bullet s^{-1}$. The plot of motions found in Figure 65 is revealing on the conditions that create motion cueing errors on rotational motions. As was the case for the heave channel, overshooting can also be found in the rotational channel. In this case it is observed that overshooting has a dampening effect on the correct motion cues sent to the motion platform. Indeed, if two opposing commands are sent to the motion platform within a short time interval, the commands cancel each other and the motion platform remains still. In this case, the overshoot cannot be attributed to the filter equation as it is of the 1^{st} order. These overshoots are caused by the fact that turn-entry and turn-exit rotational velocities typically exhibit higher frequency on their falling edge. This is a normal phenomenon that is usually dampened by the tilt-coordination channel. In the absence of sufficient tilt-coordination, this phenomenon becomes a motion cueing problem. The amplitude of simulated angular rates seems to be slightly low as it visually hovers around 20-40% of turn profile angular rates (for a scaling of *0.5*). This suggests that scaling coefficients should be increased on this channel. As such, the rotational motion channel scaling coefficients are increased for the next test run.

Having evaluated the optimal configuration of the classical WF algorithm as per the literature, an improved configuration is formulated for the second test run. This formulation is

based on pilot observations combined with an analysis of response variables and relevant motion plots. Details of this new formulation and its performance are available in Section 5.3.4.

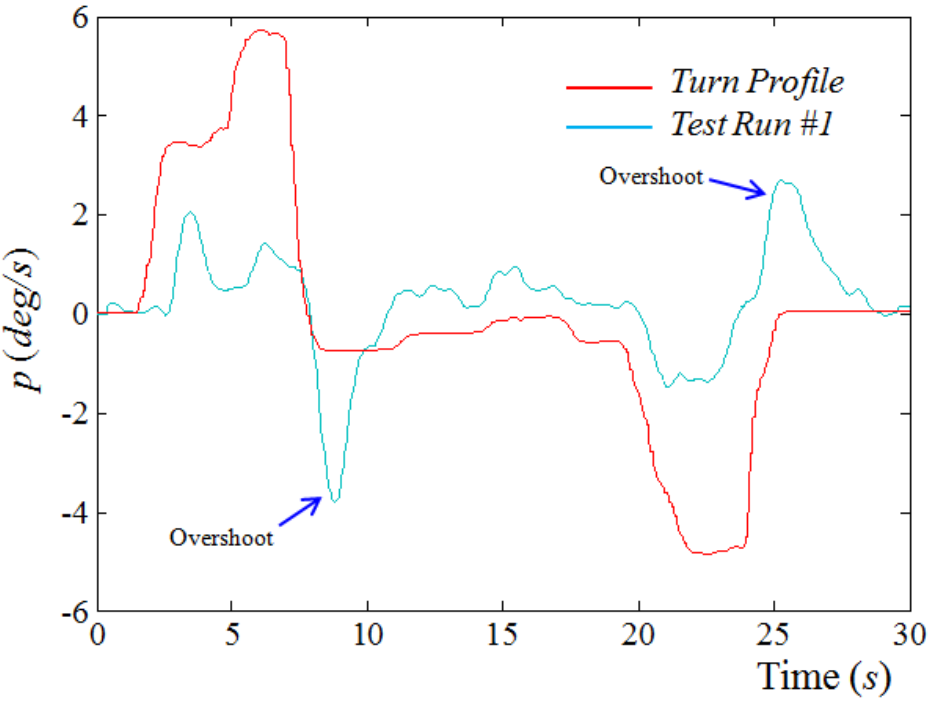


Figure 65 - *p* Angular Rates (Turn Profile vs Test Run #1)

5.3.3 Solving the Problem of 2nd Order Filter Overshooting

Through testing of different filter dampening ratio and scaling coefficients with 2nd order high pass filter equations, it was found that overshooting can never be totally eliminated. What is specifically referred to 2nd order filter overshooting can be seen in Figure 66 where unfiltered and filtered signal are compared, and where overshooting is present on almost all filtered signal peaks. At first a method using a switch based on the derivative of unfiltered signals as a criterion to zero filtered signals was investigated. The first approach consisted in monitoring the unfiltered signal derivative as criteria to zero the filtered signal. The hypothesis is that when the unfiltered signal derivative is within a certain interval of zero, no filtered signals should be let through. This makes sense because signals with a very shallow slope (with a derivative $\cong 0$) are of too

low frequency to be simulated and represent areas where there should be no motion in the filtered signal.

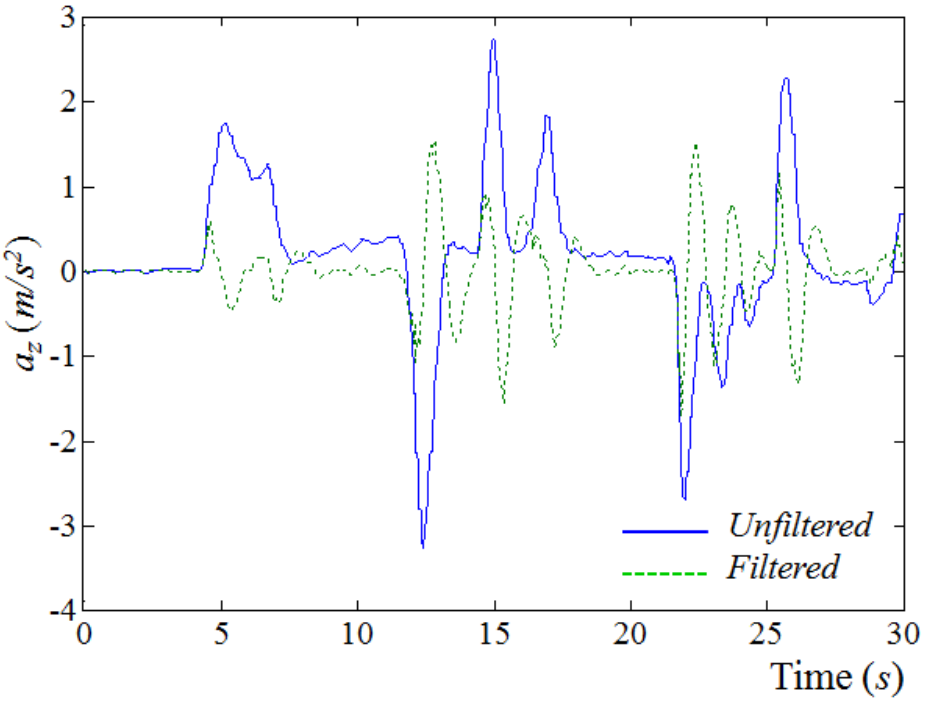


Figure 66 – 2nd Order High Pass Filter Overshooting

Thus, where shallow unfiltered slopes are present there should be less or no overshoot allowed. After a series of test runs, it was discovered that this technique is only partially effective as most overshoot begins in the middle of an acceleration peak. Because of this, they benefit from a steep slope that lets the overshoot signal permeate through to the filtered signal. Another shortfall of the derivative switch technique is that unfiltered signals are not perfect and exhibit numerous areas where the signal's slope is exactly zero. This can be seen in Figure 67 where an interval of the signal exhibiting a slope of zero is magnified. These areas of flat slope exist when a new data packet is not received within the time step interval. It is so because the algorithm is configured to output the last received value when no new value is available.

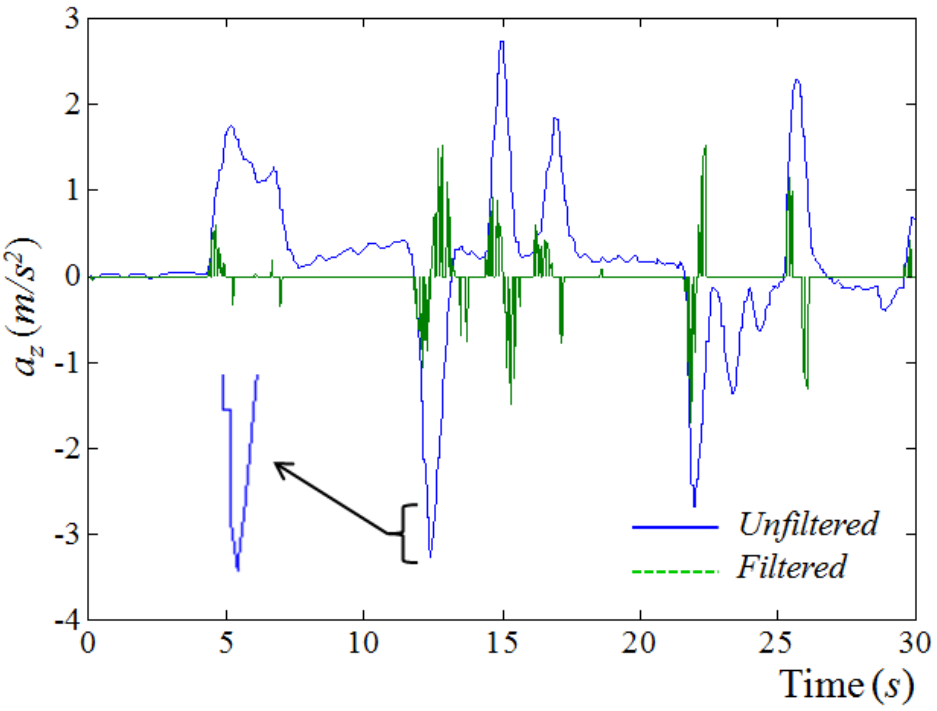


Figure 67 - Overshoot Management Using a Derivative Switch

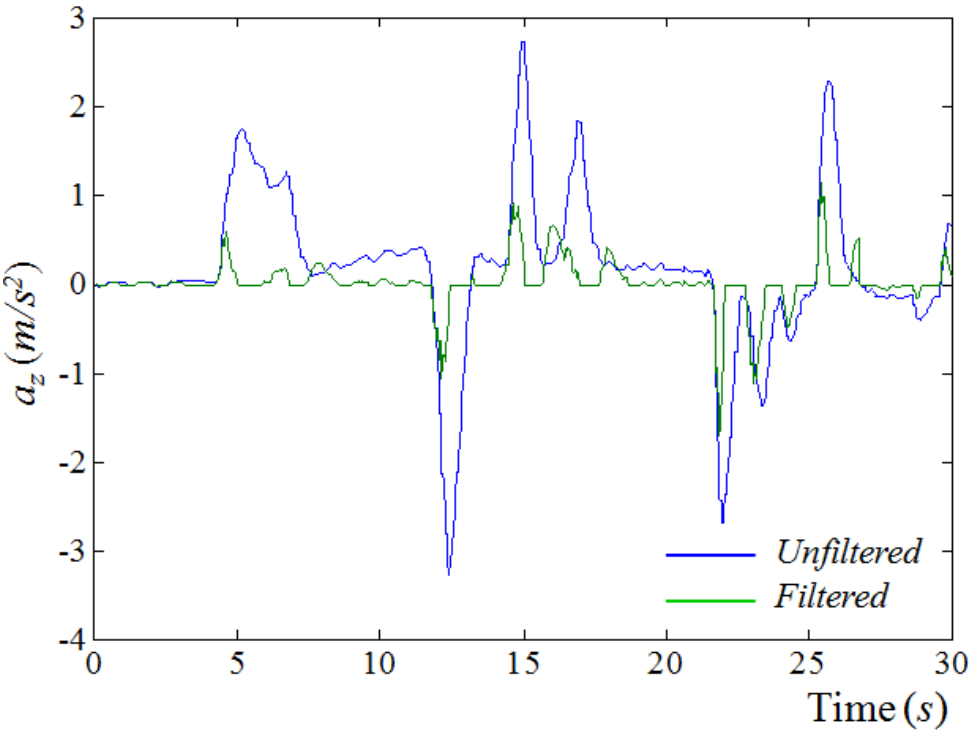


Figure 68 – Overshoot Management Using a Polarity Switch

This phenomenon causes a very noisy and underpowered signal that hops between zero and peak values when using derivative based switching. An example of this technique where filtered signals are switched based on the derivative of unfiltered signals is shown in Figure 67. In light of the problems encountered when using the derivative, another method was designed and tested to remove overshoots. The method in question is based on the parity of the unfiltered signal. Indeed, it was observed that most undesirable overshoots found in the filtered signal exhibits an inverse polarity to that of the unfiltered signal. As such, by comparing the polarity of the unfiltered signal with the filtered signal, and by switching the filtered signal to zero when polarities are different, most undesirable overshoot are neutralized. A test run featuring this overshoot management technique can be seen in Figure 68. As can be seen, this technique is very effective and reliable in most cases. This can be seen when comparing Figure 66 with Figure 68, where a definitive improvement is noted. The only shortfall of this technique is that sometimes it will fail when the unfiltered signal wanders and acquires the same polarity of an overshoot. Fortunately, it is seldom the case where both the overshoot and the unfiltered signal have the same polarity. Equation 5.5 describes the conditional statement required to implement this switch on the filtered signal. Essentially, the signal is automatically zeroed every time the polarity of unfiltered and filtered is different.

$$Filtered = \begin{cases} Filtered & If\ SGN(Filtered) == SGN(Unfiltered) \\ 0 & Else \end{cases} \quad (5.5)$$

After having studied the phenomenon of overshooting on 2nd order high pass filter, a technique using the polarity of unfiltered signals is used to remove most overshooting from filtered signals. This technique is implemented in Matlab and will be tested in the second test run.

5.3.4 Test Run #2

The second test run featured several modifications to the coefficients of the classical WF algorithm. In addition, some modifications unique to this implementation were introduced to tackle problems identified during the first test run. The main objectives of this test run were to note any improvement on heave and tilt-coordination motion. A major improvement to heave motions was expected as a result of implementing an overshoot management segment to the algorithm. A study of the tilt-coordination algorithm also led to some changes that are expected to yield a better simulation.

After several investigative test runs, it was determined that the cause of underpowered signals in the tilt-coordination channel was the tilt rate limiting block of the algorithm. Indeed, to limit the rate of tilt-coordination from an angle of tilt, the derivative is taken to obtain the tilt rate. The integral of limited tilt rate is then taken to obtain the final tilt-coordination angle. As it is often the case, the process of taking the derivative of a discrete signal is associated with heavy noise and zero crossings. In this case, signal sampling is discontinuous at a rate of 50 Hz and zero crossings can be found in typical test runs where values plateau momentarily at certain levels. A sample plot featuring the derivative (tilt rate) taken in the tilt limiting block can be seen in Figure 69. It was discovered that the tilt-coordination limiting block would scale down the tilt angle signals of the channel. This effect can be attributed to the taking of the integral when the derivative is equal to zero. In order to solve this problem while keeping the essential limiting block of the algorithm intact, a gain coefficient was introduced after the tilt limiting block.

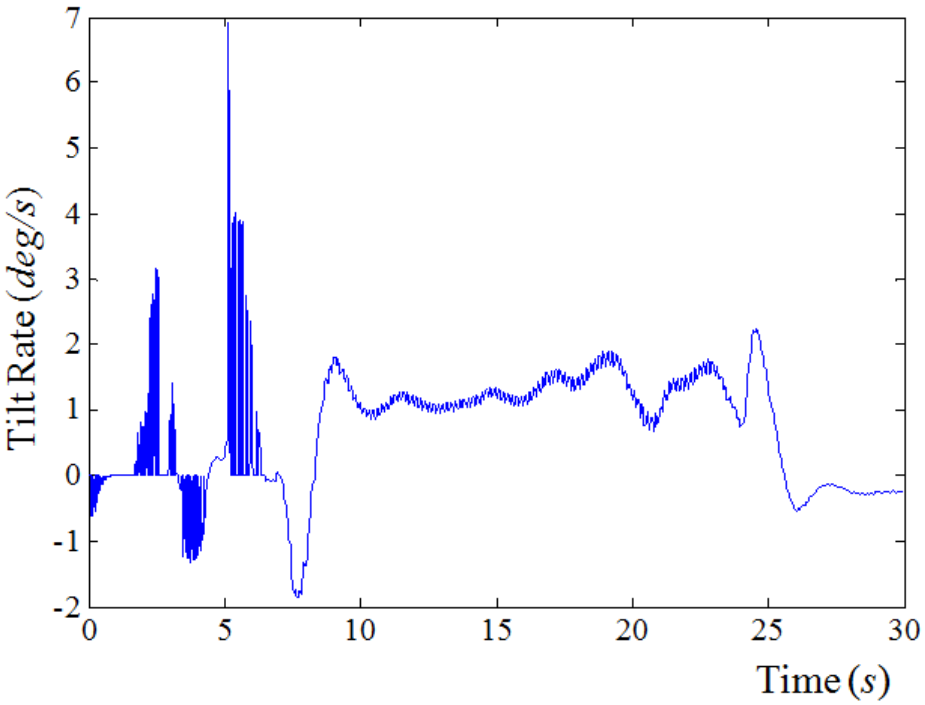


Figure 69- The effect of taking the Derivative in the Tilt Limiting Block

This gain is unique to this implementation of the classical WF algorithm and ensures that the tilt-coordination channel exhibits a signal sufficiently strong to move the motion platform as desired. Table 19 shows the coefficients used for the second test run where modifications are highlighted in red. Most scaling coefficients were adjusted and the filter damping ratio (ζ) of the heave channel was increased as recommended by [12] to reduce overshoot. Although an overshoot management segment was implemented, the adjustment of the filter Equation was beneficial for the case where the overshoot management segment fails. To begin the test run, the author sat in the motion simulator and flew several manoeuvres to evaluate the simulator response under various conditions. In essence, the response was felt to be much more realistic for all motion channels. A summary of the pilot observations collected for all motion channels during this test run can be found in Table 20.

Table 19 - Classical WF algorithm Configuration (Test Run #2)

Parameters	Scaling/Limiting	Filter Parameters	Misc.
a_x	*1*	$\omega_n = 5, \zeta = 1$	
a_y	*1*	$\omega_n = 8, \zeta = 1$	
a_z	*0.5*	$\omega_n = 4, \zeta = 4$	Overshooting Management
p	*0.6*	$\omega_n = 1$	
q	*0.6*	$\omega_n = 1$	
r	-	$\omega_n = 1$	
Tilt-Coordination	*5, 4.5 deg•s^{-1}*	-	Tilt-Coordination Gain

Table 20 - Pilot Observations (Test Run #2)

Motion Channel	Observations
Translational	Almost no overshooting present. The magnitude of accelerations is sufficient.
Tilt-Coordination	Tilt-coordination is present and proportional. Tilt-coordination sometimes lags slightly behind on pitch motions.
Rotational	Rotational motion is proportional for both pitch and roll motion. However, pitch motions seem to be underpowered.

The response variables for this test run can be found in Table 21. The variations of relevant response variables between test run #1 and #2, which is the information of interest can

be found in Table 22. The response variables of this test run were subtracted from those of the first test run. As such, a positive variation indicates that there has been an improvement on the motion response for a specific channel.

Table 21 - Response Variables for Each Motion Channels (Test Run #2)

Flight Profile	Response Variables for Each Motion Channels Relevance: Translational, Tilt-Coordination, Angular					
	a_x $(m \bullet s^{-2})$	a_y $(m \bullet s^{-2})$	a_z $(m \bullet s^{-2})$	p $(deg \bullet s^{-1})$	q $(deg \bullet s^{-1})$	r $(deg \bullet s^{-1})$
Continuous Climb	**0.7149**	0.0589	**0.2820**	0.3775	**2.0873**	0.2130
Standard Rate Turn	0.4322	**2.1833**	0.7500	**4.6425**	1.9378	5.0277
Descent	**0.7376**	0.0608	**0.3730**	0.2275	**3.0498**	0.2469

First, a definitive improvement can be found in the tilt-coordination channels. As is expected, this improvement on tilt-coordination had a slightly detrimental effect on angular channels. This is because rotational cues are generated when the platform is conducting tilt-coordination to simulate low frequency translational accelerations. Next, there is minimal variation on the *z* axis translational motion channel. However, from the pilot's perspective the improvement is quite significant due to the absence of unsolicited overshooting motions. As can be seen in the plot of *z* axis accelerations for the second test run featured in Figure 70, the signal is substantially dampened compared to that of the first test run. It is so because the filter damping ratio (ζ) of this channel's filter equation was increased while the channel's scaling coefficient remained at 0.5.

Table 22 - Response Variable Variations between Tests Run #1 and #2

Flight Profile	Response Variable Variations for Each Motion Channels Relevance: Translational, Tilt-Coordination, Angular					
	a_x $(m \bullet s^{-2})$	a_y $(m \bullet s^{-2})$	a_z $(m \bullet s^{-2})$	p $(deg \bullet s^{-1})$	q $(deg \bullet s^{-1})$	r $(deg \bullet s^{-1})$
Continuous Climb	**0.6187**	-	**0.0009**	-	**-0.5051**	-
Standard Rate Turn	-	**1.8618**	-	**-0.1899**	-	-
Descent	**0.8006**	-	**-0.0148**	-	**-1.1905**	-

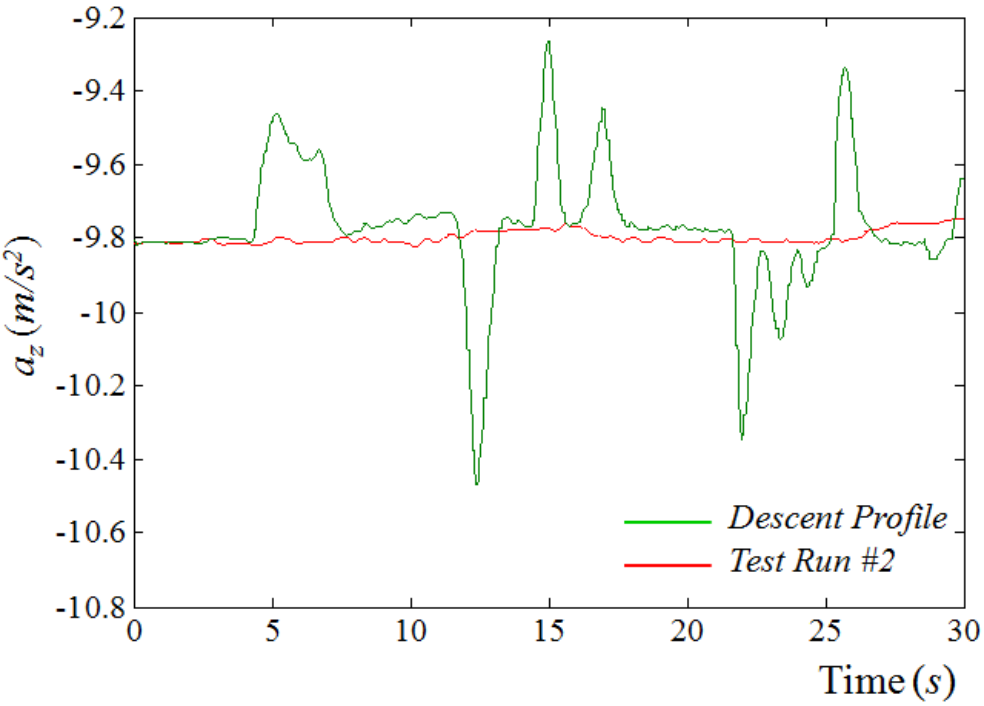


Figure 70 - z Axis Accelerations (Climb Profile vs Test Run #2)

Despite the seemingly low amplitude of the a_z signal on this particular test manoeuvre, the sensation is quite realistic from the pilot's standpoint. It is important to keep in mind that the accelerations simulated in this test manoeuvre only vary by a few tenth of $m \bullet s^{-2}$. The next test

run will feature a slightly higher scaling coefficient to investigate if more fidelity can be achieved on the heave motion channel.

Next, the tilt-coordination channel is studied for a slightly slow and underpowered tilt-coordination on pitch attitudes.

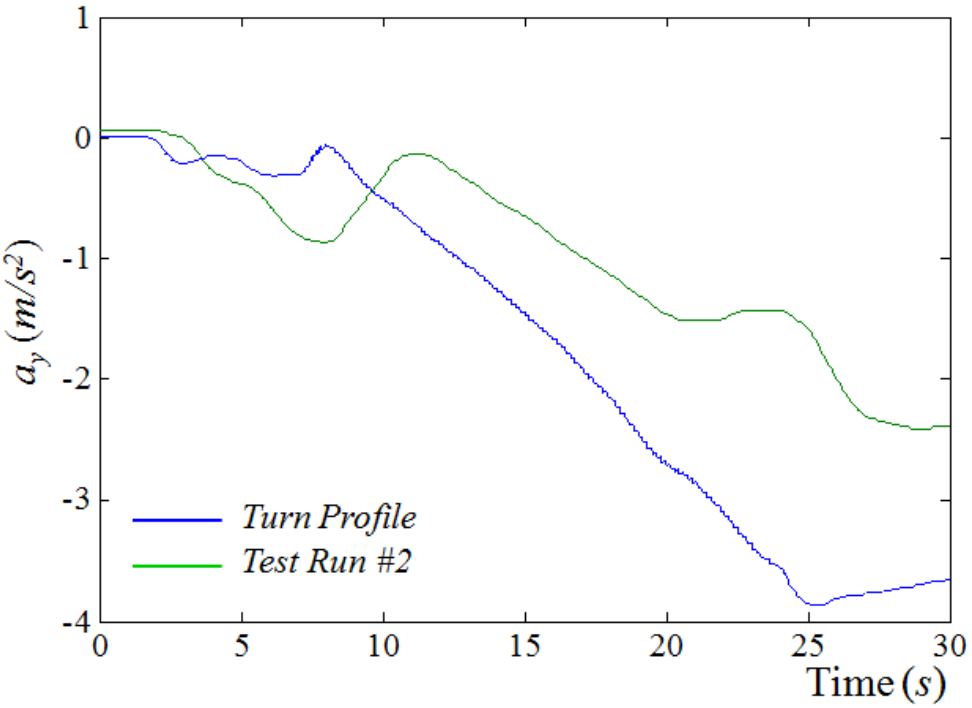


Figure 71 - *y* Axis Accelerations (Turn Profile vs Test Run #2)

By observing the variation of response variables in Table 22, it is found that gains on the *x* axis for tilt coordination are smaller than on the *y* axis. It is so because motions of heave will typically compete with motions of pitch. Indeed, pitch and heave tend to occur simultaneously as the plane is climbing or descending. It is not the case for motions of roll, which typically occurs independently of *z* axis accelerations in level flight. Consequently, additional tuning is required for the *x* axis tilt-coordination as there are interactions between motions of heave and pitch. Figure 71 shows the *y* axis accelerations for this test run, where it can be seen that the simulated translational accelerations are similar to that of the flight manoeuvre. The next test run will

feature a slightly higher tilt-rate limiting gain to ensure that pitch attitudes are felt by the pilot at a level proportional to those of roll attitudes.

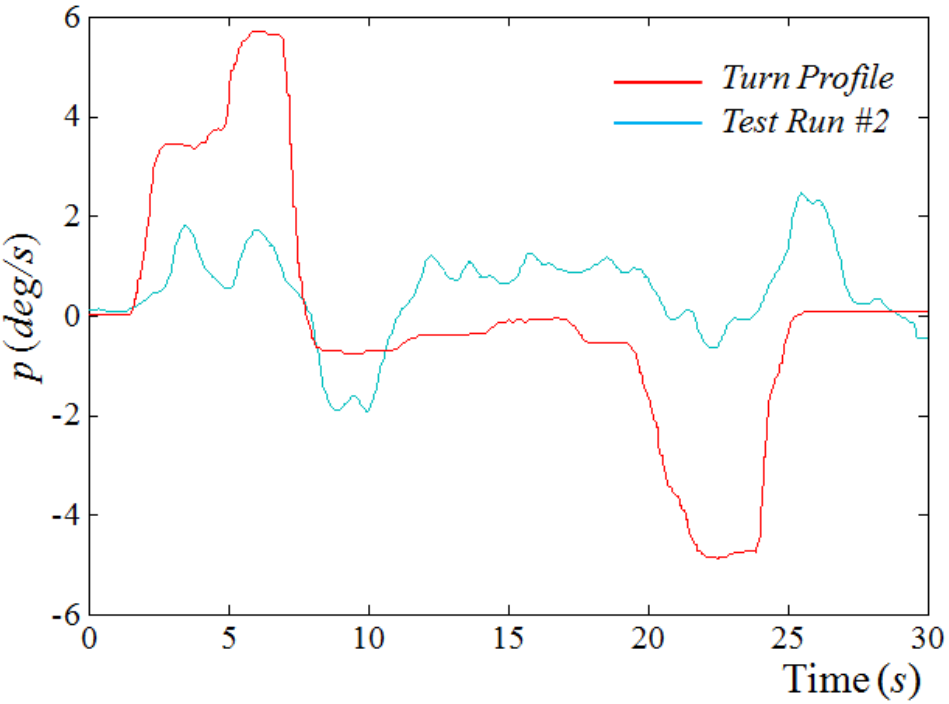


Figure 72 - *p* Angular Rates (Turn Profile vs Test Run #1)

Finally, the underpowered feeling experienced on the angular motion channel is studied. This problem is closely related to the pitch tilt-coordination problem where interactions with the heave motion channel create a dampening effect on pitch motion. For the pilot, a feeling of underpowered pitch is created by comparing the motion response of the flight simulator to roll manoeuvres with pitch manoeuvres. As such, it is critical to ensure that both roll and pitch motions create a sensation of the same magnitude for equivalent rotations about the *x* and *y* axis. In light of these observations, the next test run will feature an increased scaling coefficient on the *y* axis angular motion channel.

The second run featured many improvements to the motions generated during flight. Having identified the remaining causes of pilot complaints, the next test run will feature a configuration intended to create an optimal degree of fidelity.

5.3.5 Test Run #3

The configuration of the third test run can be found in Table 23 where changes from the second test run are highlighted in red. The main objective of this test run is to attempt to gain fidelity on pitch motions, which were identified as underpowered in the previous run.

Table 23 - Classical WF algorithm Configuration (Test Run #3)

Parameters	Scaling/Limiting	Filter Parameters	Misc.
a_x	*1*	$\omega_n = 5, \zeta = 1$	
a_y	*1*	$\omega_n = 8, \zeta = 1$	
a_z	*0.55*	$\omega_n = 4, \zeta = 4$	Overshooting Management
p	*0.6*	$\omega_n = 1$	
q	*0.7*	$\omega_n = 1$	
r	-	$\omega_n = 1$	
Tilt-Coordination	*4.5 deg•s^{-1}*	-	Gain x = *5*, Gain y = 6

As per the protocol, the author flew several flight manoeuvres to study the motion response of the algorithm. A summary of pilot observations can be found in Table 24. Overall, it is felt that an increase of the scaling coefficient for *x* axis angular rates and tilt-coordination had a positive effect on the motion generation.

Table 24 - Pilot Observations (Test Run #3)

Motion Channel	Observations
Translational	*Z* axis accelerations are not too powerful and contribute nicely to the overall simulation.
Tilt-Coordination	Tilt-coordination is present and proportional.
Rotational	Rotational motion is proportional for both pitch and roll motion. Pitch motions are in tune with *z* axis accelerations.

The response variables were calculated and compared to those of the second test run in order to objectively corroborate the improvement noted by pilot testing. It was found that the variance of response variables was very small for most motion channel. It was also observed that where gains of performance were present, a decrease in performance was also found on related channels. These observations indicate that the system is reaching its limits were improvements on channels are no longer possible without an associated loss on a related channel. It is felt that the current solution is well balanced between each motion channel and that the final result is quite realistic. Table 25 shows the response variable variations between the second and third test run. As can be seen from Table 25, the only major gain of performance can be found on the *x* axis tilt-coordination channel (a_x channel) for the continuous climb manoeuvre. This gain of performance is associated with a loss of performance on pitch motion (q channel) for this

manoeuvre. Other response variables of this test run remain relatively stable with slight variations.

Table 25 - Response Variable Variations between Tests Run #2 and #3

Flight Profile	Response Variable Variations for Each Motion Channels Relevance: Translational, Tilt-Coordination, Angular					
	a_x $(m \bullet s^{-2})$	a_y $(m \bullet s^{-2})$	a_z $(m \bullet s^{-2})$	p $(deg \bullet s^{-1})$	q $(deg \bullet s^{-1})$	r $(deg \bullet s^{-1})$
Continuous Climb	**0.1970**	-	**-0.0006**	-	**-0.1576**	-
Standard Rate Turn	-	**0.0777**	-	**-0.0724**	-	-
Descent	**-0.0220**	-	**-0.0002**	-	**0.0546**	-

In light of the results collected during this test run, it is felt that the system reached an optimal level of tuning and that the experimental study of the system reached a point where additional iteration of the experimental method would not yield substantial improvements. In addition, as pilot observations indicated that all motion channels create a realistic and proportional sensation, there is no reason to modify any channels further.

5.3.5 Conclusion

Several problems were identified during the experimental test runs. The combined method of pilot observation, response variable measurement and plot investigation used in this study was effective at identifying the problems of the classical WF algorithm. In most cases, solutions were quickly found to correct pilot complaints. The experimental test runs led to the modification of the classical WF algorithm in two areas. First, a switching mechanism based on the polarity of unfiltered accelerations was implemented to manage 2nd order filter overshooting. The results of this implementation had a positive effect on the overall motion simulation. Next, a

gain coefficient was implemented to solve the problem of an underpowered tilt-coordination. This implementation also led to a positive effect on the overall motion simulation. An iterative process was followed to fine tune the classical WF algorithm and led to a solution which is felt to be very realistic. During this process, positive improvements were also tracked on relevant motion channels between each test runs by calculating variations on response variables. The experimental study reached a conclusion when gains of performance were minimal for modifications to the configuration and where improvements on select motion channels were no longer possible without disproportional lost of performance on related channels. To provide an impartial measure of the flight simulation, a pilot validation exercise was then conducted using the configuration derived from the experimental study of the motion response.

5.4 Pilot Validation Exercise

After having optimally tuned the classical WF algorithm through an experimental observation of the system, a pilot validation exercise was conducted to assess the perceived pilot performance of the flight simulator. In addition, this exercise was used to collect pilot ratings on the unique design of RMC's 3DOF flight simulator. This validation exercise was seen as an opportunity to collect pilot ratings and subjective performance data. Comments were also collected during the pilot validation phase. Such comments will prove to be valuable for future projects involving aircraft simulation and RMC's 3DOF motion platform. In the section to follow, the activities conducted with volunteer pilots will be discussed. Finally, the pilot ratings obtained during this exercise will be given.

5.4.1 Procedure

Several pilots with various degrees of experience were approached with regards to the flight simulator and their possible participation in a pilot rating exercise. A procedure for the conduct of the pilot validation exercise was developed and will be discussed in this section. The protocol and nature of this validation exercise were approved by the Research Ethics Board of RMC. An informed consent to proceed with the validation exercise was collected from each participant. The Research Ethics Board proposal and approved protocol information can be found in Annex J.

All volunteer pilots were first introduced to the flight simulator through a short briefing highlighting the main features of RMC's 3DOF flight simulator. Then the conduct of the pilot validation exercise was discussed.

Table 26 - Pilot Validation Exercise Procedure

Step	Description
1	Participant arrival: Introduction to the flight simulator and introduction to the pilot validation procedure.
2	Briefing: Explanation of the rating questionnaire as well the manoeuvres used for rating.
3	Initial run: The participant performs a circuit without motion to get used to the controls, the airplane and the manoeuvres.
4	Validation run: The participant performs the same circuit with full motion to evaluate the flight simulator.
5	Rating: The participant fills the questionnaire to rate the flight simulator.
6	Debriefing: The participant is debriefed and additional feedback is collected trough a discussion.

More specifically, the rating criteria were explained to the pilots prior to performing the simulation. This information is covered to ensure that pilots have an idea of the important criteria for the evaluation during their flights. Table 26 summarizes the procedure which was followed for each test pilot during the pilot validation phase. The evaluation featured the conduct of an airport circuit flight manoeuvre with and without motion simulation. The pilot validation exercise focused on collecting pilot ratings on three topics. First, the pilots were asked to rate unique aspects of hardware used on RMC's 3DOF flight simulator (i.e. the controls used, the head tracking, etc). A scale from 1 to 5 (1 representing the most positive rating and 5 representing the worst rating) was used to rate each element of the simulator's hardware. Then, the pilots are asked to rate the quality of the motion based on seven criteria developed at the UTIAS by [4] during a pilot validation exercise. These criteria were selected as they are used in the literature to characterize motion quality and because they are in line with the information of interest for this study. Then, the pilots were asked to rate the impact of motion simulation by rating their perceived value of motion simulation by comparing the motionless run to the full motion one. Finally, the pilots were asked to comment and rate the potential for using RMC's 3DOF as a training aid for pilots under training. The pilot rating questionnaire that was used to collect the pilot ratings can be found in Annex H.

Volunteers were directed to observe silence during the tests when other pilots were present. This measure was taken to prevent any discussion of the simulator's performance, which could lead to a bias in the group responses. The focus was put on collecting individual and unique evaluation from each pilot.

5.4.2 Pilot Ratings and Comments

A total of 12 pilots with varying degrees of experience conducted the pilot evaluation exercise. The following section will discuss the pilot ratings and comments obtained during the validation exercise.

The pilot validation exercise benefited from a wide range of pilots with different qualifications. Pilot flight time ranged from 85 hours to more than 10 000 hours. Qualifications ranged from private pilot to fully qualified, multi-engine, commercial instructors. It was also possible to obtain the input of a military test pilot who had previous experience in simulator evaluation. Relevant information such as pilot flight time, qualifications and aircraft flown can be found in Annex I.

The ratings collected through the pilot rating questionnaire were separated into four categories relating to hardware, motion, flight simulation impact and training potential. Table 27 shows the average ratings collected for the flight simulator hardware assessment. The complete results for each pilot can be found in Annex I.

Table 27 - Average Pilot Ratings for Flight Simulation Hardware

Pilot	Visual	Head Tracking	Controls	Audio	Cockpit	Overall
Average	2.17	2.21	2.21	1.38	1.63	1.69
Deviation	0.81	0.96	1.05	0.48	0.71	0.51

* Ratings from 1 to 5, where 1 is excellent and where 5 very poor.

Overall, pilots deemed the hardware used to be adequate for the conduct of flight simulation. As reflected by the average ratings and standard deviations, pilots consistently rated the audio, cockpit and overall attributes with very positive ratings. Higher average ratings and

larger standard deviations between pilot ratings for the visual, head tracking and controls attributes was noted. These also represent the areas where pilot had the most complaints and suggestions.

Table 28 – Summary of Pilot Complaints for the Flight Simulator Hardware

Equipment	Complaints
Head mounted display	Pilots with prescription glasses had a very hard time with the head mounted display. Some pilots complained about the resolution of the display. Some pilots found it hard to adjust the head mounted display. Some pilots reported slight discomfort as the head mounted display rested on their nose.
Head tracking	Some pilots were dissatisfied by the gain on head motions, especially in the yaw channel. Pilots felt a learning period was required make the best use of the head tracking. Some weird visual orientations were reported when pilots reached the limits of head tracking. Finally, some pilots noted a slight delay between head movement and the image displayed.
Controls	Some pilots were dissatisfied by the non-linear gain exhibited by the joystick. Some pilots were also dissatisfied by the limited amplitude of the joystick. Resistance was deemed to be slightly weak when compared to the control column of an actual aircraft. Pilots reported oversensitive for progressive flaps retraction and trim control. Most pilots complained about the absence of rudder pedals.

Pilot complaints for the flight simulator hardware where targeted at limitations exhibited by the visual, head tracking and control equipment. It was found that the bulk of pilot feedback was aimed at the hardware utilized on the motion platform rather than the sensation of motion. As such, Table 28 summarizes the complaints collected from pilots during the validation exercise. Certain complaints were more significant than others. Indeed, most pilots found the use of head tracking to be a major improvement over the use of a static display. However, most pilots were also critical of the limitations of the head tracking system in use, but agreed that this technique showed great potential if complaints were addressed. Next, the absence of rudder pedals was noted by pilots as the most detrimental to the flying experience. Pilots were dissatisfied by the inability to center the aircraft properly during landing and takeoff manoeuvres. Finally, pilots with prescription glasses had a hard time with the head mounted display. This aspect was not discussed during the design phase and highlights the fact that such a system is not suitable for all. Despite the seemingly long list of complaints and comments, it is felt that improvements can be easily achieved by purchasing new equipment or changing some simple configuration settings.

The average motion simulation pilot ratings collected during the validation exercise can be found in Table 29. As expected, the areas of least performance are found to be the smoothness and amplitude attributes of motion. All pilots experienced little discomfort or disorientation during the flight simulation.

Table 29 – Average Pilot Ratings for Motion Simulation

Pilot	Smoothness	Sense	Amplitude	Phase Lag	Discomfort	Disorientation	Overall
Average	2.67	1.92	2.27	1.90	1.21	1.50	1.81
Deviation	0.75	0.60	0.67	0.56	0.40	0.77	0.58

* Ratings from 1 to 5, where 1 is excellent and where 5 is very poor.

Overall, it was felt by pilots that the sense of motions was appropriate and without any noticeable phase lags. By comparison to the hardware evaluation, there were very little complaints with regards to the motion simulation aspect of flight simulation. Most pilots who reported unfavourable ratings for smoothness complained that the initial input to manoeuvers was slightly overpowered. Unfavorable ratings for phase lag were attributed by most pilots to an initial phase lag when entering a manoeuvre. Unfavorable ratings for the amplitude rating were most of time targeted at decelerations that were sometimes felt to be slightly overpowered, especially in turns. Overall, the motion simulation was felt to be adequate and devoid of major problems.

After having completed both a motionless and full motion run, the pilots were asked to rate the impact of motion simulation on the overall flight simulation. In addition, pilots were asked to give their opinion on the flight simulator potential as a training aid. These ratings provided interesting data on two subjective topics of great importance in defining the future of RMC's 3DOF flight simulator. The average ratings collected for both these attributes are shown in Table 30.

Table 30 – Average Pilot Ratings for Motion Impact and Flight Training Potential

Pilot	Motion Simulation Impact	Flight Training Potential
Average	1.52	1.42
Deviation	0.43	0.47

* Ratings from 1 to 5, where 1 is excellent and where 5 is very poor.

The results show that there is a strong conscensus between pilots on the positive impact of motion simulation. In addition, all pilots feel that there is a potential for training applications using a flight simulator such as RMC's 3DOF flight simulator. As part of the rating

questionnaire, pilots were asked to specifically provide comments on potential applications for RMC's 3DOF flight simulator in flight training. Table 31 summarizes the numerous suggestions and ideas which were identified by pilots for RMC's 3DOF flight simulator. Overall, there is a general consensus amongst pilots that the flight simulator could be used to perform numerous tasks that do not require a full size realistic cockpit. It is important to mention that most pilots also cautioned that the system's flaws would need to be "ironed-out" before it could be used commercially for any of those tasks. In light of these suggestions, a role could be attributed to the system and it could be redesigned and re-tuned to optimally fulfill this role.

Table 31 - Pilot Suggestions for Potential Training Use

Pilot Suggestions for Potential Training Use
To train procedures, emergency response and intrument flying. For ab initio pilot training on circuit procedure and airspeed management. To practice a flight before actually doing it for real. To train inadvertent entry in instrument meteorological conditions (used to recreate the associated disorientation). To be used as a glider simulator for Air Cadets.

5.5 Summary

A pilot validation exercise was designed to study the flight simulation system. Best practise from the previous studies involving pilots were incorporated into the procedure usd in this study. The pilots were asked to rate the flight simulator on four topics of interest. The pilot validation exercise provided critical information that helped answer some very important questions about RMC's 3DOF flight simulator. Although the pilot evaluations remain largely subjective and will vary greatly depending of pilot experience and perspective, they remain

important when trying to evaluate the fidelity of a flight simulator. Overall, pilots were satisfied with the flight simulator and there was a conscensus that motion was greatly beneficial to flight simulation. Areas of complaints emerged almost exclusively from limitation of the hardware used in this study. It is fair to say that almost all pilot complaints could be eliminated or alleviated using better equipment (which is available in most cases). From these results it is fair to say that RMC's 3DOF motion platform is suitable for use as a flight simulator. This represents the most important conclusion emerging from this study. With that in mind, the next step is to define a more ambitious project where a training application is selected and where the system is designed specifically to suit that application.

CHAPTER 6 - CLOSING REMARKS

6.1 Introduction

The work conducted in this study was aimed at designing, implementing and analysing a flight simulator that takes full advantage of RMC's 3DOF motion platform. More specifically, the challenge was to design a system using commercially available components at a very low cost that would achieve acceptable levels of performance for aircraft simulation. To obtain maximum performance from RMC's 3DOF, the underlying principles of the classical washout filter (WF) algorithm were carefully studied. The classical WF algorithm was selected due its wide use in commercial simulation, simplicity and favourable pilot response as reported in the literature. The classical WF algorithm was then modified to be suitable for 3DOF use and implemented using the Simulink environment. Microsoft's Flight Simulator X was selected as the virtual environment of choice due to its realism in all areas and its extensive software development kit. A C++ executable was developed using VS2008 and makes use of a software design pattern, the mediator pattern, in order to interface and control the flight simulator components. The entire system was designed using a design process where design objectives and constraints were studied. This led to a final system which was implemented as envisioned in the original design. Finally, an experiment was designed to fine tune the classical WF algorithm and to collect performance data. The thorough evaluation of the flight simulation capabilities of RMC's 3DOF flight simulator through an experimental study and a pilot validation exercise led to a better understanding of RMC's 3DOF motion platform as a tool to create flight simulation.

Modifications were also made to the classical implementation of the classical WF algorithm to resolve issues of 2nd order filter overshooting and underpowered tilt-coordination channel. These modifications led to a significant improvement to motion simulation. The trials conducted with qualified pilots also provided invaluable information from potential users of such a system. The positive ratings collected from pilots are an indicator that low cost motion simulation using commercially available products is a promising approach. The information collected from these trials opens the door to many possible improvements on the existing system. Overall, the work and conclusions drawn in this work will be relevant to RMC as it moves to a more advanced flight simulator using a 6DOF motion platform. This chapter will discuss the areas of interest for future work on RMC's 3DOF and the use of motion simulation at RMC. Specific areas of possible improvement for the classical WF algorithm and the flight simulator hardware are presented followed by a range of possible applications for RMC'S 3DOF motion platform in the light of the findings of this study.

6.2 Future Work

The study of RMC's 3DOF flight simulator through the experimental tuning and pilot validation processes has lead to several observations that can lead to significant improvements to the motion cueing algorithm and flight simulator hardware. The following sections will discuss these observations in light of the literature, experimental results and pilot comments, to formulate recommendations for future work on flight simulators at RMC.

6.2.1 Classical WF Algorithm Improvement

In this study, the use of an inertial measurement unit was introduced to measure the accelerations and angular rates experienced on the motion platform. However, it is believed that

such a device can play a more active role as part of the WF algorithm. Most WF algorithms assume a perfect simulator response to motion commands. In reality, this is not the case as small variations exist between the actual and assumed platform position. Because the motion platform is constantly washing out to its central position there is no troublesome error accumulation and this formulation works relatively well. In this case, the true position of the motion platform can be obtained directly from the Galil motion controller and the motions can be measured with the IMU 320. It is believed that the classical WF algorithm or some other form of WF algorithm could be modified to use this hardware-in-the-loop configuration to improve the motion simulation. Intelligent approaches such as the use of human inner ear (otholith) models as measures of performance combined with fuzzy logic and genetical algorithm as reported by [31] could be studied and modified to develop an algorithm capable of automatically tuning its filter equations from the actual motions experienced on the motion platform. The study of an acceleration feedback mechanism as in [32] shows that the classical WF algorithm can also be studied and included into hardware-in-the-loop formulations. In light of the time consuming and repetitive task of algorithm tuning, it seems that the formulation of an auto-tuning method would be a major breakthrough. Such a method, if sufficiently reliable, could also reduce the need for pilot evaluation exercises in establishing a flight simulator's performance.

6.2.2 Flight Simulator Hardware Improvement

Improvement of system performance through the acquisition of better hardware components can become a never-ending process. In the case of flight simulation, this process ends when a complete recreation of motions, audiovisuals and tactile aspects of flight has been achieved. If economical constraints are an important consideration, it will not be possible to design or acquire the advanced and costly hardware necessary to fully achieve this goal. The

comments formulated in this section reflect the observations of pilots and the author in order to improve the simulator while maintaining the low cost philosophy of this study.

As identified during the pilot testing phase, the head tracking system is perhaps one of the most important component in the system. However, due to the nature of the TrackIR 5, pilots could only move their head within a 120 degree cone in front of the sensor. This led to a non-proportional head tracking where the pilot's head would move more in the virtual world than it did in reality. According to most pilots who tested the flight simulator, improvements on head tracking would significantly increase their feeling of immersion in a real aicraft. As such, it would be fitting to look into products which would allow more freedom in head movement. Perhaps a combination of several TrackIRs with customized software would allow for more freedom. Another option would be to develop an experimental tracking system capable of all around motions at RMC. Such a system would certainly find wide usage in various virtual reality applications. Future work in this area could also include a study of the techniques used in the cinematographic industy for the tracking of actors in movies where computer generated imagery is used.

The second area of unanimous complaint from pilots arose from the lack of rudder pedals on the flight simulator. Indeed, without rudder input it became hard for pilots to achieve centerline on takeoff and landing manoeuvres. This became evident during the pilot validation phase where pilots were asked to perform circuits around the airport. Although there were no rudders on RMC's 3DOF, rudders could easily be implemented on RMC's 6DOF (features a large platform area and yaw motion). This would allow for the use of the flight simulator as a training aid for feet inputs, which are critical during circuit manoeuvres. A rudder system is already available as part of the G940 flight system used in this study.

Finally, the third area of complaints from pilots targeted the joystick controller, which was deemed to have a very limited range when compared to an actual aircraft stick. As such, the use of a joystick with a wider motion envelope could improve the flight simulator control. Such a joystick could potentially be found commercially at customized flight simulation businesses. It can be pointed out that many of these hardware limitations experienced will be removed as hardware components continue to be improved by the industry.

6.2.3 Future Applications

Although an acceptable level of flight simulation has been achieved with RMC's 3DOF, it remains an experimental flight simulator designed to study the problem of flight simulation and the possibilities associated with RMC's 3DOF motion platform. One of the next steps consists in determining a potential realistic application for this kind of flight simulator. Then, a careful design could be made to be specifically oriented to this task. As suggested by many pilots, this platform could potentially be used to train new pilots to accomplish manoeuvres before doing them in the real aircraft. Most pilots with training experience also saw great potential for such a flight simulator as an inexpensive IFR trainer. Most pilots felt that the motion experienced would contribute greatly to IFR scenarios where the visibility is inexistent. In such cases, the pilot is forced to rely on his instruments and not on the motion cues. The introduction of motion cues could train pilots to ignore them and focus on their instruments.

RMC's 3DOF flight simulator is also perfectly suited for the study of advanced algorithms using hardware-in-the-loop configurations. Its low cost and simplicity allows for the testing of ideas in short periods of time. Such ideas could include the simulation of other vehicles such as tanks and armoured personal carriers. The use of software already in use within the CF such as Virtual Battle Space 2 (VBS2) [33] could provide the virtual environment for military

vehicules or aircrafts. This would allow for the simple and seamless integration of motion simulators to a networked environment of VBS 2. In light of the low cost of RMC's 3DOF motion platform, it might be interesting and also economically feasible to network multiple simulators within this environment. In essence, with the right virtual environment it would be possible to use motion in many applications. Depending on the task, finding suitable virtual environment without building them from the ground up can be a challenging consideration when exploring the simulation of new vehicles.

ANNEXES

Annex A - Motion Platform Security Briefing

RMC's 3DOF Motion Platform Security Briefing

RMC's 3DOF motion platform is quite powerful and if used improperly could cause injuries to personnel or damage equipment. The motion platform operator is responsible for the safety of platform passengers and spectators during the operation of the motion platform. To ensure safety during platform operation the following rules must be followed at all times:

1) Before applying power to the motion platform, ensure that all personnel and equipment is at a safe distance of the motion platform.

2) When passengers are boarding and un-boarding the motion platform, it should always be in the "parked" position and powered-off using the safety switch.

3) Before the motion platform is energized, the motion platform operator must assist the passenger in safely fastening his seat harness. The passenger harness must remain fastened until the platform is completely shut-down (safety switch is off).

4) The motion simulation can be halted at any time by the passenger by pressing the "pause" button. This button must be clearly identified to the passenger by the motion platform operator.

5) The motion platform operator must provide a short familiarization briefing covering the layout of controls and other pertinent information to new passengers.

6) In the event of an **emergency**, the following procedure should be followed:

a) Attempt to stop the motion simulation using the software interface.

b) If the software fails, power-off the motion platform using the safety switch.

c) When the platform is powered-off, manage the emergency situation.

Annex B - Flight Simulator Components and Cost

Component	Manufacturer	Approx. Cost (CDN)	Role
2-Inch Harness Belts	Corbeau	$85.00	Pilot seat safety harness
A4 Racing Seat	Corbeau	$350.00	Pilot seat
Control Shaft and TrackIR 5 Mount	Woodman Machine Products	$1,216.35	Custom harness for pilot controls
Fatal1ty Gaming Headset	Creative	$55.00	Audio
Flight Simulator X	Microsoft	$40.00	Virtual flight simulation environment
Flight System G940	Logitech	$300.00	Pilot input system (controls)
Heavy Duty 60A Safety Switch	SquareD	$75.00	Motion platform safety switch
RMC's 3DOF Motion Platform	SSSI	$55,000.00	Motion cues generation
TrackIR 5 and Trackclip Pro	NaturalPoint	$175.00	6DOF Head tracking
USB Extension 32-ft	Sabrent	$18.00	Extension for TrackIR 5
USB Extension 32-ft	Sabrent	$18.00	Extension for controls
USB Extension 32-ft	Sabrent	$18.00	Extension for Z800 3DVisor
VGA Extension 25-ft	Ultra	$36.00	Extension for Z800 3DVisor
VGA Extension 25-ft	Ultra	$36.00	Extension for Z800 3DVisor
Z800 3DVisor	eMagin	$1,800.00	Visual display
	Total:	$59,222.35	

Annex C - Flight Simulator Wiring Diagram

Figure 73 is a simple diagram showing the specific layout of cables used to bring the system together. It provides additional information for anyone trying to understand the implementation of the system using the components selected during the design. Several universal serial bus (USB), video graphics array (VGA) and audio extensions were required to accomplish this task. It is interesting to note that two VGA cables are required for the visual display. It is so because the head mounted visual display acts as the secondary screen of the control station with a VGA splitter which sends a signal to the Pilot View Display (a display designed to allow spectator to see what the pilot sees). The IMU 320 is powered using a power supply which is not illustrated here. Power is supplied with the serial cable.

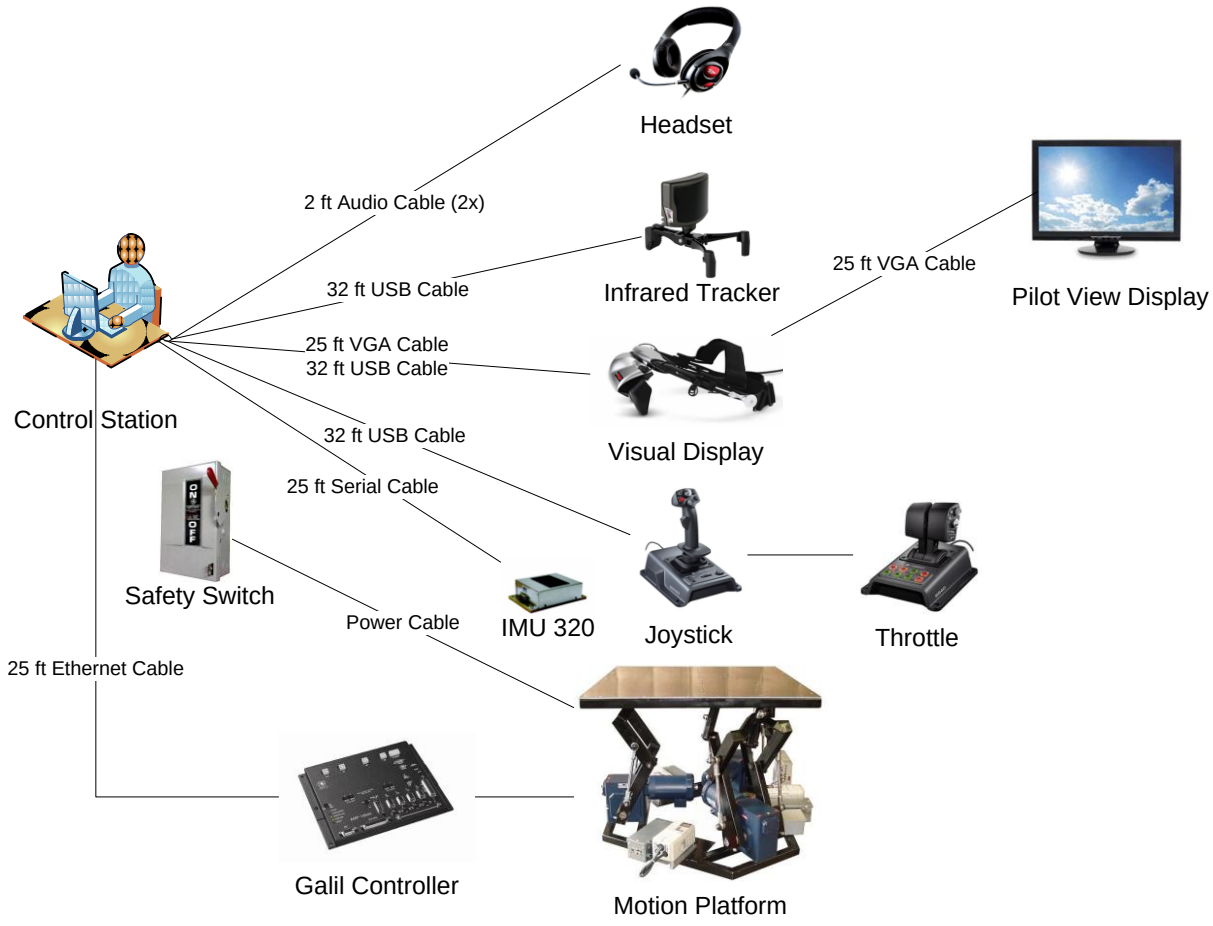
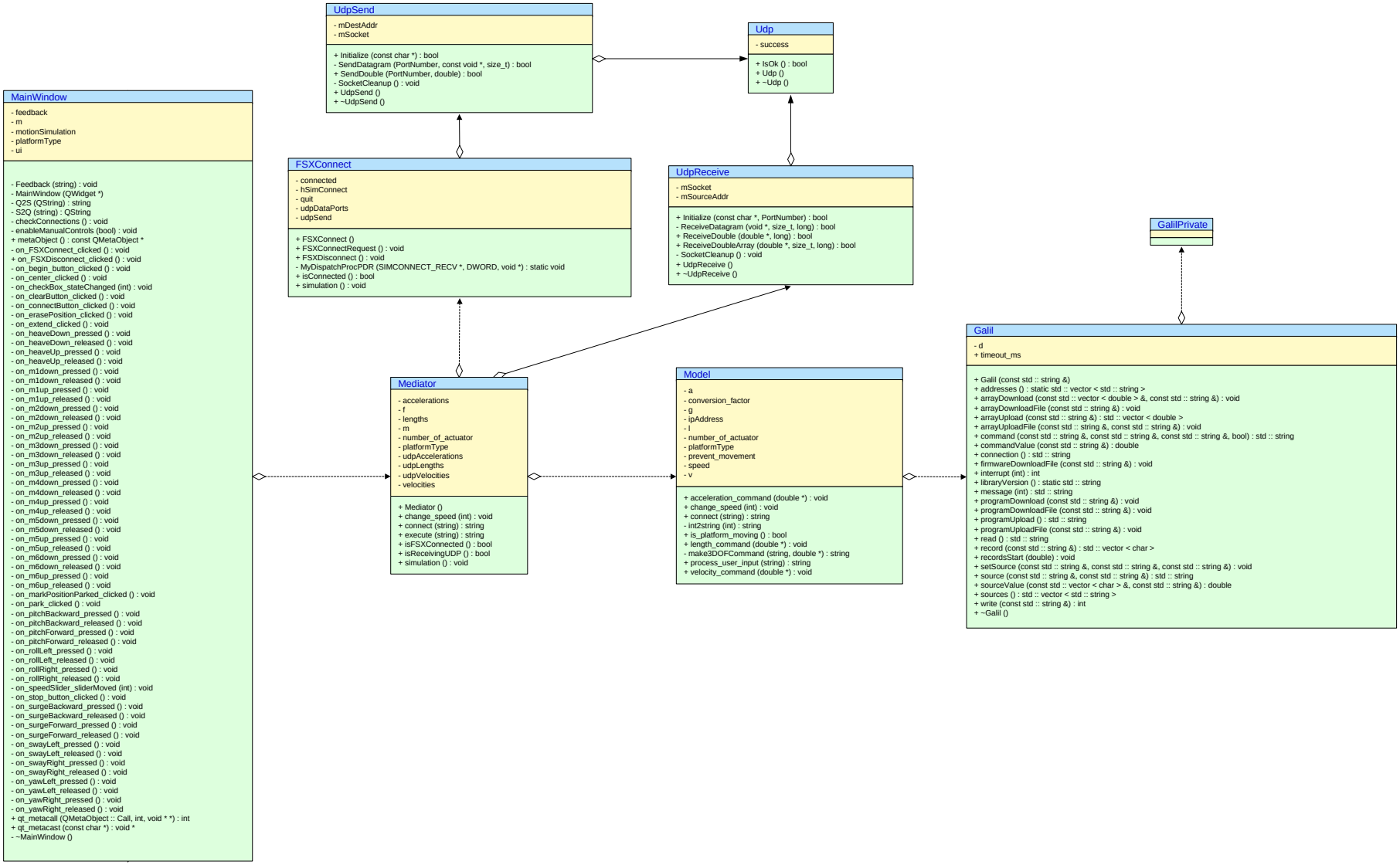


Figure 73 - Flight Simulator Wiring

Annex D - UML Class Diagram


UdpSend
- mDestAddr
- mSocket
+ Initialize (const char *) : bool
- SendDatagram (PortNumber, const void *, size_t) : bool
+ SendDouble (PortNumber, double) : bool
- SocketCleanup () : void
+ UdpSend ()
+ ~UdpSend ()
Udp
- success
+ IsOk () : bool
+ Udp ()
+ ~Udp ()
MainWindow
- feedback
- m
- motionSimulation
- platformType
- ui
- Feedback (string) : void
- MainWindow (QWidget *)
- Q2S (QString) : string
- S2Q (string) : QString
- checkConnections () : void
- enableManualControls (bool) : void
+ metaObject () : const QMetaObject *
- on_FSXConnect_clicked () : void
+ on_FSXDisconnect_clicked () : void
- on_begin_button_clicked () : void
- on_center_clicked () : void
- on_checkBox_stateChanged (int) : void
- on_clearButton_clicked () : void
- on_connectButton_clicked () : void
- on_erasePosition_clicked () : void
- on_extend_clicked () : void
- on_heaveDown_pressed () : void
- on_heaveDown_released () : void
- on_heaveUp_pressed () : void
- on_heaveUp_released () : void
- on_m1down_pressed () : void
- on_m1down_released () : void
- on_m1up_pressed () : void
- on_m1up_released () : void
- on_m2down_pressed () : void
- on_m2down_released () : void
- on_m2up_pressed () : void
- on_m2up_released () : void
- on_m3down_pressed () : void
- on_m3down_released () : void
- on_m3up_pressed () : void
- on_m3up_released () : void
- on_m4down_pressed () : void
- on_m4down_released () : void
- on_m4up_pressed () : void
- on_m4up_released () : void
- on_m5down_pressed () : void
- on_m5down_released () : void
- on_m5up_pressed () : void
- on_m5up_released () : void
- on_m6down_pressed () : void
- on_m6down_released () : void
- on_m6up_pressed () : void
- on_m6up_released () : void
- on_markPositionParked_clicked () : void
- on_park_clicked () : void
- on_pitchBackward_pressed () : void
- on_pitchBackward_released () : void
- on_pitchForward_pressed () : void
- on_pitchForward_released () : void
- on_rollLeft_pressed () : void
- on_rollLeft_released () : void
- on_rollRight_pressed () : void
- on_rollRight_released () : void
- on_speedSlider_sliderMoved (int) : void
- on_stop_button_clicked () : void
- on_surgeBackward_pressed () : void
- on_surgeBackward_released () : void
- on_surgeForward_pressed () : void
- on_surgeForward_released () : void
- on_swayLeft_pressed () : void
- on_swayLeft_released () : void
- on_swayRight_pressed () : void
- on_swayRight_released () : void
- on_yawLeft_pressed () : void
- on_yawLeft_released () : void
- on_yawRight_pressed () : void
- on_yawRight_released () : void
+ qt_metacall (QMetaObject :: Call, int, void * *) : int
+ qt_metacast (const char *) : void *
- ~MainWindow ()
FSXConnect
- connected
- hSimConnect
- quit
- udpDataPorts
- udpSend
+ FSXConnect ()
+ FSXConnectRequest () : void
+ FSXDisconnect () : void
- MyDispatchProcPDR (SIMCONNECT_RECV *, DWORD, void *) : static void
+ isConnected () : bool
+ simulation () : void
UdpReceive
- mSocket
- mSourceAddr
+ Initialize (const char *, PortNumber) : bool
- ReceiveDatagram (void *, size_t, long) : bool
+ ReceiveDouble (double *, long) : bool
+ ReceiveDoubleArray (double *, size_t, long) : bool
- SocketCleanup () : void
+ UdpReceive ()
+ ~UdpReceive ()
GalilPrivate
Mediator
- accelerations
- f
- lengths
- m
- number_of_actuator
- platformType
- udpAccelerations
- udpLengths
- udpVelocities
- velocities
+ Mediator ()
+ change_speed (int) : void
+ connect (string) : string
+ execute (string) : string
+ isFSXConnected () : bool
+ isReceivingUDP () : bool
+ simulation () : void
Model
- a
- conversion_factor
- g
- ipAddress
- l
- number_of_actuator
- platformType
- prevent_movement
- speed
- v
+ acceleration_command (double *) : void
+ change_speed (int) : void
+ connect (string) : string
- int2string (int) : string
+ is_platform_moving () : bool
+ length_command (double *) : void
- make3DOFCommand (string, double *) : string
+ process_user_input (string) : string
+ velocity_command (double *) : void
Galil
- d
+ timeout_ms
+ Galil (const std :: string &)
+ addresses () : static std :: vector < std :: string >
+ arrayDownload (const std :: vector < double > &, const std :: string &) : void
+ arrayDownloadFile (const std :: string &) : void
+ arrayUpload (const std :: string &) : std :: vector < double >
+ arrayUploadFile (const std :: string &, const std :: string &) : void
+ command (const std :: string &, const std :: string &, const std :: string &, bool) : std :: string
+ commandValue (const std :: string &) : double
+ connection () : std :: string
+ firmwareDownloadFile (const std :: string &) : void
+ interrupt (int) : int
+ libraryVersion () : static std :: string
+ message (int) : std :: string
+ programDownload (const std :: string &) : void
+ programDownloadFile (const std :: string &) : void
+ programUpload () : std :: string
+ programUploadFile (const std :: string &) : void
+ read () : std :: string
+ record (const std :: string &) : std :: vector < char >
+ recordsStart (double) : void
+ setSource (const std :: string &, const std :: string &, const std :: string &) : void
+ source (const std :: string &, const std :: string &) : std :: string
+ sourceValue (const std :: vector < char > &, const std :: string &) : double
+ sources () : std :: vector < std :: string >
+ write (const std :: string &) : int
+ ~Galil ()

Annex E - Response Variable Calculation Code

The calculation of response variables between test runs is a repetitive task. As such a MATLAB function (CalculateRV) was developed to expedite the experimental process. This code performs the smoothing of recorded signals and returns the response variable for each motion channel. The code found on the MATLAB exchange to perform the triangular smoothing technique used for smoothing the signals is also included.

```
function RV = CalculateRV(Dataset, FlightProfile)
%   Function: CalculateRV
%
%   Calulates the Response Variables and standard deviation.
%   Dataset represents the recorded motions of a test run.
%   FlightProfile represents the reference flight profile.
%
%   Author: 2Lt Jean Le Bouthillier
%   Date: September 5 2010

%   First the effect of gravity will be added to the Flight profile data as
%   it is seperate from specific accelerations
len =  length(FlightProfile.signals.values);
FlightProfile.signals.values(1:len,1:3) = FlightProfile.signals.values(1:len,1:3) + FlightProfile.signals.values(1:len,4:6);
FlightProfile.signals.values(1:len,4:6) = FlightProfile.signals.values(1:len,7:9);

%   We are saving the first and last 10 values from the z axis force channel.
%   This is necessary to prevent the smoothing algorithm from zeroing those
%   values. Their average is calculated and added to the smoothed dataset.

%   Smooth the Crossbow Acceleration signals
%   Ax
head = mean(Dataset.signals.values(1:10, 1));
tail = mean(Dataset.signals.values(len-10:len, 1));
Dataset.signals.values(1:len,1) = fastsmooth(Dataset.signals.values(1:len,1), 10);
Dataset.signals.values(1:10, 1) = head;
Dataset.signals.values(len-10:len, 1) = tail;
%   Ay
head = mean(Dataset.signals.values(1:10, 2));
tail = mean(Dataset.signals.values(len-10:len, 2));
Dataset.signals.values(1:len,2) = fastsmooth(Dataset.signals.values(1:len,2), 10);
Dataset.signals.values(1:10, 2) = head;
Dataset.signals.values(len-10:len, 2) = tail;
%   Az
head = mean(Dataset.signals.values(1:10, 3));
tail = mean(Dataset.signals.values(len-10:len, 3));
Dataset.signals.values(1:len,3) = fastsmooth(Dataset.signals.values(1:len,3), 10);
Dataset.signals.values(1:10, 3) = head;
Dataset.signals.values(len-10:len, 3) = tail;

%   Smooth the remaining Angular rate signals
%   p
head = mean(Dataset.signals.values(1:10, 4));
tail = mean(Dataset.signals.values(len-10:len, 4));
Dataset.signals.values(1:len,4) = fastsmooth(Dataset.signals.values(1:len,4), 10);
Dataset.signals.values(1:10, 4) = head;
Dataset.signals.values(len-10:len, 4) = tail;
%   q
```

```
head = mean(Dataset.signals.values(1:10, 5));
tail = mean(Dataset.signals.values(len-10:len, 5));
Dataset.signals.values(1:len,5) = fastsmooth(Dataset.signals.values(1:len,5), 10);
Dataset.signals.values(1:10, 5) = head;
Dataset.signals.values(len-10:len, 5) = tail;
%   r
head = mean(Dataset.signals.values(1:10, 6));
tail = mean(Dataset.signals.values(len-10:len, 6));
Dataset.signals.values(1:len,6) = fastsmooth(Dataset.signals.values(1:len,6), 10);
Dataset.signals.values(1:10, 6) = head;
Dataset.signals.values(len-10:len, 6) = tail;

%   Now the Response variables are calculated for each motion channels
RV = mean(abs(FlightProfile.signals.values(1:len,1:6) - Dataset.signals.values(1:len,1:6)));
%   Now calculate the standard deviation for each response variables
STD = std(abs(FlightProfile.signals.values(1:len,1:6) - Dataset.signals.values(1:len,1:6)));
RV = RV + (2*STD);
%   A plot of reference flight profile and dataset is printed.
plot(Dataset.time, Dataset.signals.values(1:len,1:6), FlightProfile.time, FlightProfile.signals.values(1:len, 1:6))
end

function SmoothY=fastsmooth(Y,smoothwidth)
%   fastsmooth(Y,w) smooths vector Y by triangular
%   smooth of width = smoothwidth. Works well with signals up to
%   100,000 points in length and smooth widths up to 1000 points.
%   Faster than tsmooth for smooth widths above 600 points.
%   T. C. O'Haver, 2006.
w=round(smoothwidth);
SumPoints=sum(Y(1:w));
s=zeros(size(Y));
halfw=round(w/2);
for k=1:length(Y)-w,
  s(k+halfw)=SumPoints;
  SumPoints=SumPoints-Y(k);
  SumPoints=SumPoints+Y(k+w);
end
s=s./w;
SumPoints=sum(s(1:w));
SmoothY=zeros(size(s));
for k=1:length(s)-w,
  SmoothY(k+halfw)=SumPoints;
  SumPoints=SumPoints-s(k);
  SumPoints=SumPoints+s(k+w);
end
SmoothY=SmoothY./w;
```

Annex F - Crossbow IMU 320 Simulink Interface

The Crossbow IMU 320 was interfaced with the Simulink environment in order to collect accelerations and angular rates from RMC's 3DOF flight simulator. The IMU comes with a user's manual [26] which contains a programming guide including all the necessary information for interfacing the unit. Many data packets can be sent and received from the IMU. For this study, the packet of interest (the S1 packet) contains the accelerations, angular rates and temperature information. As can be seen from Figure 74, all Crossbow IMU 320 packets conform to a specific packet format and are always headed by the bit pattern 0x5555.

Packet Header 0x5555	*Packet Type (2 Bytes)* Unsigned Short	*Payload Length (1 Byte)* Unsigned Char	*Payload (Variable Length)* Variable Format	*CRC (2 Bytes)* Unsigned Short

Figure 74- Crossbow IMU 320 Packet Format

For the S1 packet, the Packet Type identifier is 0x5331 (S1) and the payload lenght is 18 bytes. The accelerations and angular rates are received first in the payload as signed short with 2's complement and take up a total of 12 bytes of storage. As can be seen in Figure 75, the Serial receive block of the Instrument Control Toolbox by Mathworks is used to listen on the serial port for the incoming IMU packets. The configurations as seen in Figure 76 are set in accordance with Crossbow's specifications for serial communication with the IMU 320. The Serial Receive block is configured to receive packet with a header of 0x55555331 (UUS1). Then, only the following 13 bytes of data (the payload lenght and the relevant 12 bytes of data) are kept. A simple process is applied to transform the bytes into relevant acceleration and angular rates. First, the most significant byte of each variable is multiplied by 256 for it to gain its true value. In this case, the values are transmitted using the Big Endian standard which means the most significant byte is first. Then, both bytes are added together to form a double value (the output type is set in

the add block parameters). Finally, a scaling factor is applied to the values as per Crossbow's specifications. In this case the accelerations are scaled by a factor of $\frac{20}{2^{16}}$ and the angular rates are scaled by a factor of $\frac{7\pi}{2^{16}}$.

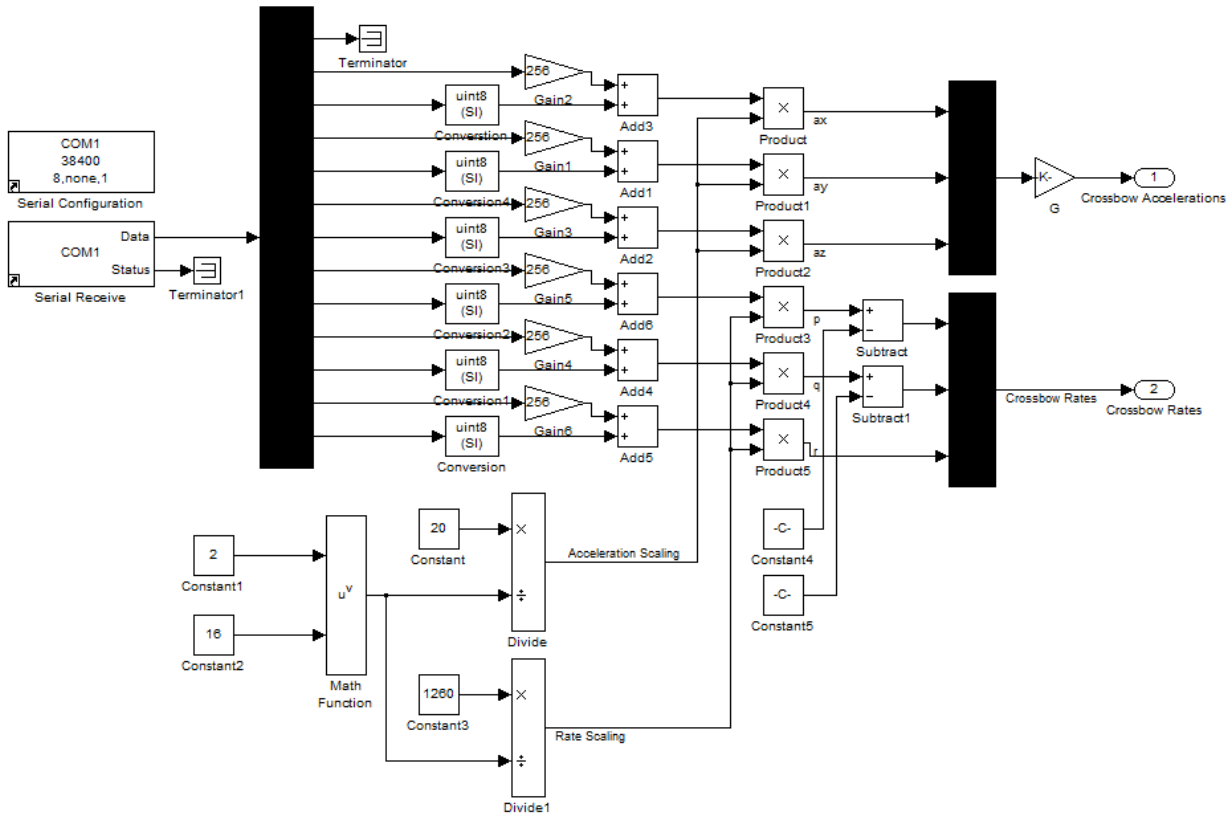


Figure 75 - Crossbow IMU 320 Simulink Interface

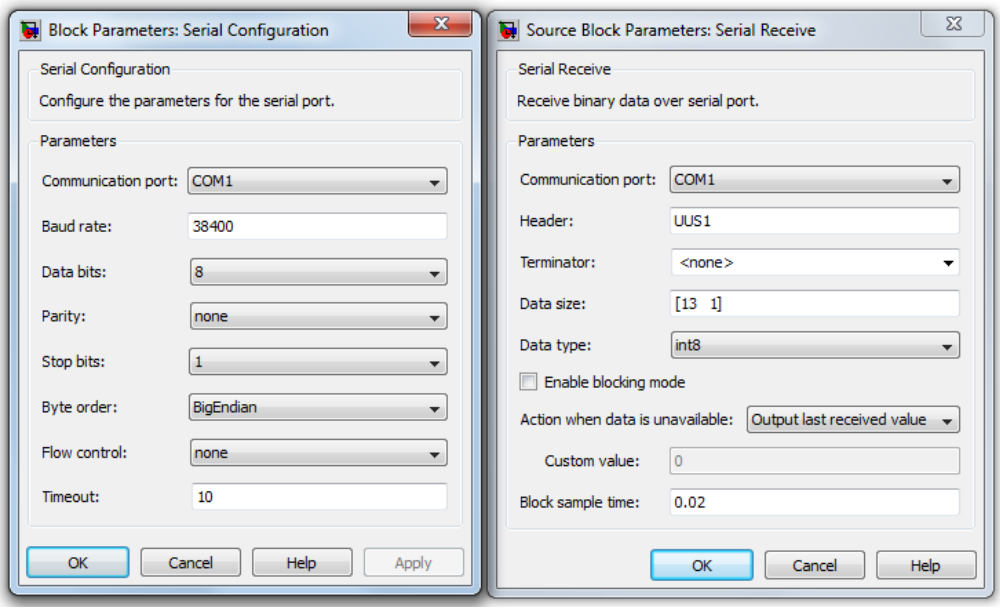


Figure 76 - Simulink Serial Configurations (IMU 320)

As can be seen in Figure 75, two constants are substracted from the *x* and *y* angular velocity channels. It is so because the particular unit used for this study exhibits a slight signal deviation from the origin (even when the IMU is motionless). Those constants were established by recording the running average of each rotational velocity signal for a long period. Using these constants, all three rotational velocity channel exhibit a signal equal to zero when the IMU is motionless.

Annex G - Recording and Playback

The implementation of the classical WF algorithm was modified twice to perform recording and playback operations. For recording operations, a flight recorder block was simply added to the classical WF algorithm implementation. Using this block, the motions were recorded to the MATLAB workspace as a “Structure With Time” featuring a double decimation and recording at the same frequency as the system. Figure 77 shows the top level view of the Fligth Recorder block and Figure 78 shows the inside view of the same block.

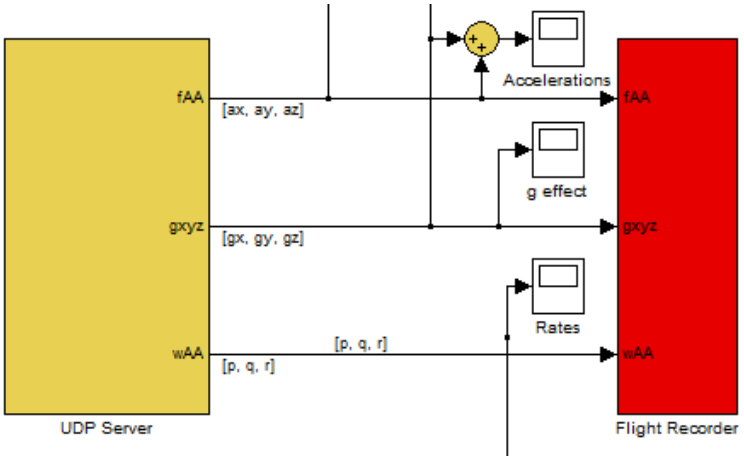


Figure 77 - Top Level View of Flight Recorder Block

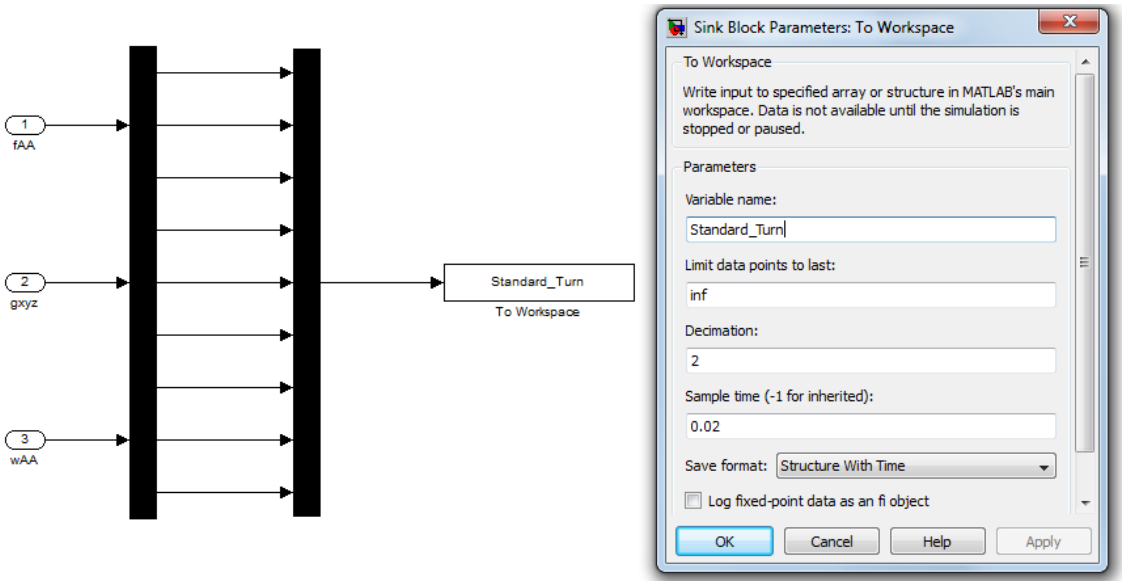


Figure 78 - Inside View of Flight Recording Block

The playback functionality was implemented by replacing the UDP Server block by the Flight Playback block which is configured to read “Structure With Time” from the MATLAB environment. This block feeds into the unchanged components of the classical WF algorithm implementation to drive the motion platform. Finally, a Crossbow IMU recording block was added to the model in order to record the platform motions into the MATLAB environment as a “Structure With Time”. Inside the Crossbow Recorder block, information is recorded in a similar fashion than for the Flight Recorder block of Figure 78.

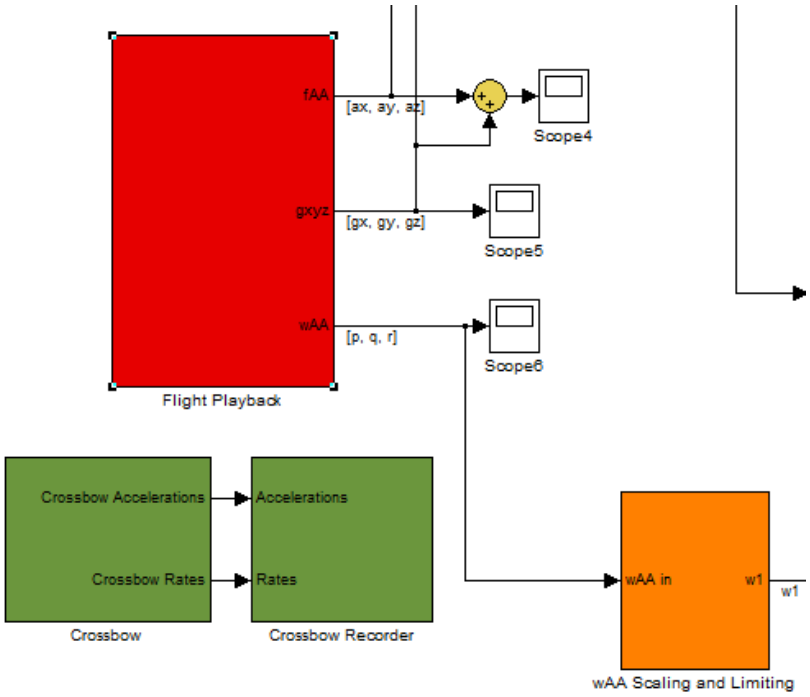


Figure 79 - Top Level View of Playback and Crossbow Blocks

An inside view of the Flight Playback block where a “From Workspace” block is used to fetch data at the frequency of the system can be seen in Figure 80. The data interpolation option is selected to ensure that a value is always outputted from the Flight Playback block. The block is also configured to output zero after the final data value is read from “Structure With Time”.

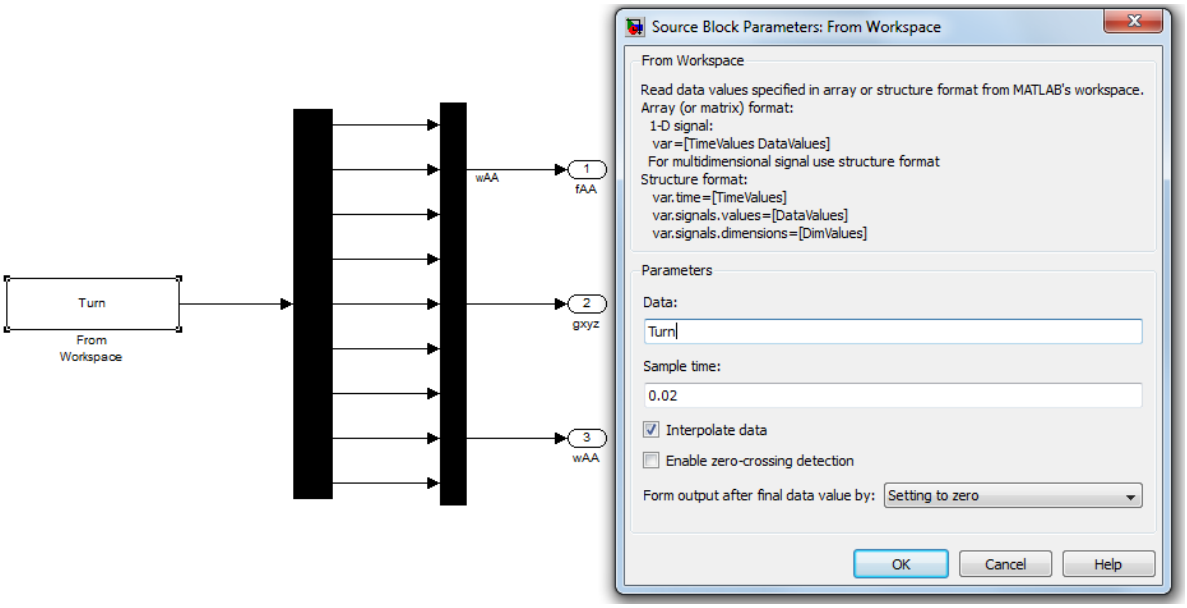


Figure 80 - Inside View of Flight Playback block

Annex H - Pilot Rating Questionnaire

RMC's 3DOF Flight Simulator Evaluation

Pilot Information

Please answer the following questions about your flying experience.

1) What is your total flight time? _____ **hrs**

2) Please list the main aircraft(s) on which you have acquired this flight time:

_____

3) Please list your pilot qualifications (i.e. private, commercial, instructor etc.):

_____

4) Have you ever flown the Cessna 172? **Yes / No**

If so, how many hours do you have on the C-172? _____ **hrs**

Flight Simulation Hardware

Please indicate your assessment of the simulation hardware based on the following rating scales:

Attribute	***Rating***			***Response***
Visual:	Excellent - creates a very realistic feeling	1----------5	Unrealistic feeling - detrimental to the experience	
Head tracking:	Excellent - creates a very realistic feeling	1----------5	Unrealistic feeling - detrimental to the experience	
Controls:	Excellent response to inputs – provides excellent control	1----------5	Very poor control - detrimental to the experience	
Audio:	Excellent - creates a very realistic feeling	1----------5	Unrealistic feeling - detrimental to the experience	
Cockpit:	Ergonomic – very comfortable	1----------5	Uncomfortable, detrimental to the experience	
Overall:	Excellent	1----------5	Extremely poor	

Motion Simulation Rating

Please indicate your assessment of motion simulation based on the following rating scales:

Attribute	***Rating***			***Response***
Smoothness:	Extremely smooth - comparable with fixed base	1----------5	Extremely jerky - limit of tolerance	
Sense:	Definitely correct as in aircraft	1----------5	Totally reversed	
Amplitude:	No motion experienced	1----------5	At least twice that expected	
Phase Lag:	None experienced	1----------5	At least 180°	
Discomfort:	None experienced	1----------5	Cannot continue manoeuvre	
Disorientation:	None experienced	1----------5	Cannot perform Manoeuvre	
Overall:	Excellent	1----------5	Extremely poor	

Motion Simulation Impact

Please indicate your assessment of motion simulation impact based on the following rating scale

Attribute	***Rating***			***Response***
Impact:	Motion simulation has a very positive impact on the flight simulation	1----------5	Motion simulation has no impact or is even detrimental to flight simulation	

Flight Training Potential

If you have experience with flight instruction:

Do you see a potential use for this kind of flight simulator as a flight instruction aid? Please elaborate.

Please include a rating from 1 to 5 on the training potential of RMC's 3DOF flight simulator.

Attribute	***Rating***			***Response***
Potential:	RMC's 3DOF flight simulator has great potential for flight training	1----------5	RMC's 3DOF flight simulator has no potential for flight training	

Additional Comments

Feel free to enter any comments with regards to the flight simulation experience:

Annex I - Pilot Information and Ratings

Table 32 - Pilot Information

Pilot	Total Flight Time (hrs)	C-172 Flight Time (hrs)	Experience	Aircrafts Flown
1	10000	3000	Commercial, Instructor, Multi Engine, IFR	Various commercial aircrafts up to Boeing 737
2	2600	None	Military helicopter pilot, Standards officer, Test pilot	CH146, CH136, CH118
3	800	100	Commercial, Instructor, Multi Engine, IFR	C-150, C-172, Zlin-242
4	798.9	500	Commercial, Multi Engine, IFR	C-172, PA-28-140, Piper Seminole, Beech Duchess
5	103.9	50	Private, Primary flight training (Air Force)	C-150, C-172, Grob-120A
6	110	50	Private	C-152, C-172
7	100	70	Glider, Private	SGS233A, C-172
8	400	75	Commercial, Instructor	C-150, C-172
9	280	150	Commercial, Instructor	C-150, C-152, C-172
10	225	90	Glider, Private	C-150, C-152, C-172, Katana DV-20
11	85	55	Private, Primary flight training (Air Force)	C-172, Grob-120A
12	260	70	Private, Primary flight training (Air Force)	C-150, C-152, C172, Grob-120A

* Pilot flight time is in most cases approximate as reported by the pilots on the evaluation questionnaire.

Table 33 - Complete Hardware Ratings

Pilot	Visual	Head Tracking	Controls	Audio	Cockpit	Overall
1	1.5	1.5	2	1	1	1.25
2	3	4.5	3.5	2	2	3
3	2	3	2.5	2	3	2
4	4	2	1.5	1	1	1.5
5	2	2	2	1	2	2
6	2	1	2	2	2	1
7	2	2	1.5	1	1	1.5
8	3	3	5	2	2	2
9	1	2	1.5	1.5	2.5	1.5
10	2	2.5	1.5	1	1	1.5
11	1.5	1	2	1	1	1.5
12	2	2	1.5	1	1	1.5
Average	2.17	2.21	2.21	1.38	1.63	1.69
Deviation	0.81	0.96	1.05	0.48	0.71	0.51

* Ratings from 1 to 5, where 1 is excellent and where 5 is very poor.

Table 34 - Complete Motion Simulation Ratings

Pilot	Smoothness	Sense	Amplitude	Phase Lag	Discomfort	Disorientation	Overall
1	2	1.5	2	1.75	1.5	1	1.75
2	2.5	2	2	3	1	1.5	2
3	2.5	2	2	2.5	1	1	2
4	3.5	1.5	2.5	1.5	1	1	1.5
5	2.5	2	2	2.5	1	1	2
6	2	3	2	2	2	2	2
7	2.5	2	3	1.5	1	1.5	1
8	4	2	4	1	2	3	3
9	2	1	2	2	1	1	1.5
10	2	1.5	1.5	1.5	1	1	1
11	4	3	2.5	1.5	1	3	2.5
12	2.5	1.5	1.7	2	1	1	1.5
Average	2.67	1.92	2.27	1.90	1.21	1.50	1.81
Deviation	0.75	0.60	0.67	0.56	0.40	0.77	0.58

* Ratings from 1 to 5, where 1 is excellent and where 5 is very poor.

Table 35 - Complete Impact and Training Potential Ratings

Pilot	Motion Simulation Impact	Flight Training Potential
1	1.75	1
2	2	2
3	2	1
4	1.5	2
5	1.5	2
6	1	1
7	1	1
8	2	2
9	1.5	1.5
10	1	1.5
11	2	1
12	1	1
Average	1.52	1.42
Deviation	0.43	0.47

* Ratings from 1 to 5, where 1 is excellent and where 5 is very poor.

Annex J – Ethics Board Documentation

The conduct of a pilot validation exercise with human participation required the authorization from RMC's Research Ethics board. In this annex, the submission for ethics review, the informed consent letter and final authorization letter are included.

SUBMISSION FOR ETHICS REVIEW OF
RESEARCH INVOLVING HUMAN SUBJECTS

REMINDER:

- **If you are conducting research in an institution/agency other than RMC, you may be required to get ethical approval from other boards before proceeding with your research. Approval may also be required for the area in which the research will occur (e.g., Nunavut, Indian reserves, international research). Please ensure that you have all required approval prior to conducting your project.**

- **Investigators and co-investigators must ensure that all personnel involved in their research projects are aware of, and adhere to the ethical principles outlined in the Tri-Council Policy statement.**

- **For bilingual projects, all relevant documentation must be presented in both languages.**

- **You must obtain RMC REB approval even if REB approval has been given at another institution. You may be required to complete this application; please contact the chair for advice.**

- **Any questions regarding the necessity for ethics review should be directed to the RMC REB Chair.**

- **You may need DHRRE approval if you or your human subjects fall outside of RMC jurisdiction. Further information can be obtained from the Chair of RMC REB.**

- **Please submit one electronic copy or three hard copies of the submission to the Chair or to the office of the Research Services.**

- **The research project cannot proceed until ethics approval is obtained from the RMC REB.**

A. GENERAL INFORMATION

1. Title of Project:

The Design, Implementation and Study of a Low Cost 3 Degree-of-Freedom Flight Simulator using RMC'S 3 DOF Motion Platform

Indicate the anticipated commencement date for this project: 25 October 2010

Indicate the anticipated completion date for this project: 05 November 2010

2. Applicant	**Dept:**	**Ext.:**	**Email:**
2Lt Jean Le Bouthillier	Mathematics and Computer Science	613-583-8778	jean.le.bouthillier@rmc.ca

Applicant's status:	**Faculty Supervisor** -----
Graduate Student	Dr. Yawei Liang

3. Co-Investigators/	**Dept:**	**Ext .:**	**E-mail:**

4. Level of Project:

- ☐ Faculty Research
- ☒ Graduate Research
- ☐ Undergraduate Research
- ☐ Other : specify _____

5. Funding Status:

Is this project currently funded? Yes ☒ No ☐

If **Yes,** provide: Funding Source RMC / DRDC Period of Funding: Until December 2010

If **No**, is funding being sought? Yes ☐ No ☐

Funding Source(s) _____ Period of Funding: _____

6. Has this application been submitted to another institution's Research Ethics Board?

Yes ☐ No ☒

If **Yes**, provide the name of the institution, date of ethics review and the decision. Attach a copy of the ethics clearance certificate, if applicable. _____

B. SUMMARY OF PROPOSED RESEARCH

1. Purpose and Rationale for Proposed Research

Briefly describe the objectives of the proposed project and the hypothesis(es)/research questions to be investigated. Include the potential benefits and the rationale for the study or studies. Include a copy of a research proposal (2 pages maximum), where applicable.

The objective of the study is to determine the level of flight simulation fidelity which can be achieved by using RMC's 3 degree-of-freedom (DOF) motion platform as a flight simulator (a low cost system). Traditionally, expensive 6 DOF motion platforms are utilized for flight simulation. However, as evidenced by [1], 3 DOF motion platforms exhibit potential in producing acceptable levels of motion simulation. It is theorized that the use of a 3 DOF motion platform can greatly reduce the cost of flight simulation while providing fidelity levels that are conducive to its use as a flight training aid within the Canadian Forces.

Flight simulator fidelity is achieved through a combination of software and hardware. Some aspects of flight simulation fidelity can be studied through empirical evaluation of the system using sensors and other measurements [2]. However, these evaluation techniques have their limitations and can never replace the input of qualified pilots in evaluating subjective aspects of flight simulation (i.e. control response, motion sense, visual quality, etc.). As reported by Reid and Nahon [3] from the University of Toronto Institute for Aerospace (UTIAS), pilot evaluation is essential in obtaining the full picture in evaluating the fidelity of flight simulation.

More specifically, this study aims to collect pilot feedback on four topics: the hardware utilized on RMC's 3 DOF flight simulator, the motion experienced, the impact of motion simulation and the potential applications for the flight simulator.

The motivation of this study is to acquire and publish relevant data on the use of low cost 3 DOF flight simulation systems for use in the Canadian Forces. Agencies such as Defense Research Development Canada (DRDC) Toronto and the Directorate of Land Synthetic Environment (DLSE) have already shown interest for the results of this study.

Summary information on volunteer pilots (i.e. flight time, flight qualification, aircraft flown, etc.) would be collected in order to describe the sample obtained and to study any possible impact of pilot qualifications on the data. Pilot ratings would be collected anonymously as pilot personal information is of no value for the evaluation. A rating questionnaire developed at the UTIAS and modified at RMC will be used to collect pilot ratings.

References

[1] Pouliot, N.A., Gosselin, C.M., Nahon, M.A., "Motion Simulation Capabilities of Three-Degree-of-Freedom Flight Simulators", Journal of Aircraft, Vol. 35, No. 1, January 1998, pp. 9-17

[2] Reid, L., Nahon, M., "Simulator Motion Drive Algorithms – A Designer's Perspective", Journal of Guidance, Control, and Dynamics, Vol. 13, No. 2, 1990, pp. 356-362.

[3] Reid, L.D., Nahon, M.A., "Response of Airline Pilots to Variations in Flight Simulator Motion Algorithms", Journal of Aircraft, Vol. 25, No. 7, July 1988

2. Methodology/Procedures

Provide a brief, sequential description of the procedures to be used with human participants in this study. Include a description of measures that will be used (e.g. surveys, questionnaires, interviews, focus groups, audiotaping, videotaping, unobtrusive observations, etc.). Attach a copy of all instruments and assessment tools or provide sample items if complete tools are unavailable.

Participants will be able to voluntarily choose to participate in the study by arranging an appointment (at their convenience) with the research staff. A questionnaire (Annex A) will be used to collect ratings immediately after the conduct of test runs. Then, a short discussion will be held to collect additional comments and thank the participants for their time. The only record kept for a session is the rating questionnaire.

The conduct of a session is summarized in the following table:

Step	Description
1	Participant arrival: Introduction to the flight simulator and introduction to the pilot validation procedure.
2	Briefing: Explanation of the rating questionnaire as well the manoeuvres used for rating.
3	Initial run: The participant performs a circuit without motion to get used to the controls, the airplane and the manoeuvres.
4	Validation run: The participant performs the same circuit with full motion to evaluate the flight simulator.
5	Rating: The participant fills the questionnaire to rate the flight simulator.
6	Debriefing: The participant is thanked and additional feedback is collected trough a short discussion.

3. Research Setting

If procedures involve direct contact with participants, describe the setting in which the research will take place. Indicate who will be present during the process. Describe any known or foreseeable risks for the researcher, or others involved in collecting the research data. What safeguards are in place to protect against these risks?

In person contact with participants will be made during the pilot rating exercise. Staff will welcome the participants and guide them throughout the exercise. Staff will also be responsible for operating the flight simulator.

The exercise will take place in the Girouard-333 laboratory where the flight simulator is located. This is where flight simulation will take place and where the participants will be asked to fill the rating questionnaire.

To prevent the risk of influencing the ratings and opinion of participants during the evaluation, staff will not discuss or give their opinion on the fidelity of the flight simulator. Participants will be asked to complete the rating questionnaire before discussion the experience with staff.

To prevent a consensus when multiple participants are present for one session, the participants will be asked to wait until all participants have completed the questionnaire to discuss the rating.

4. Participants Involved in the Study

a. Describe any special characteristics of potential participants in this study (e.g., race, gender, mental and physical disabilities, age range). Are there participant exclusion and/or inclusion criteria? If so, describe and provide justification.

Participants in this study will be individual with flight experience of various levels.

There will be no restriction on age or gender. The study will be conducted in English.

b. How many participants are expected to be involved in this study?

Around 10 participants are expected to be involved.

c. From what source will the potential participants be recruited?

RMC undergraduate or graduate classes ☐

Other RMC sources ☒ (specify)

→RMC Flying Club

DND military personnel ☒

DND civilian personnel ☒

Canadian Government personnel ☐

School Boards ☐

Correctional Services ☐

Agencies ☐

Mailing Lists ☐

Businesses, industries, professions ☐

Health care settings, long term care facilities, etc. ☐

Other (specify) ☐ _____

d. Describe how and by whom the potential participants will be recruited. Attach a copy of any materials to be used for recruitment (e.g., posters, flyers, advertisement, letter, telephone script).

Members of the RMC Flying club will be asked to participate. DND employees and DND civilian personnel known to have flight experience will also be asked to participate. Contact will be made in person, by phone or by email in all cases.

No formal material will be used for contacting participants. However, solicitation will always include the following information:
1) Description of the study
2) Tasks to be conducted by the participants
3) Time involved to complete the tasks

C. POTENTIAL BENEFITS TO PARTICIPANTS

1. If participants will receive any reward (financial or other) for participation, provide details:

n/a

2. Describe any other known or anticipated direct benefits to the participants from their involvement in the project.

Participants will have the opportunity to fly a realistic flight simulator featuring motion simulation and head tracking. Interested participants will be able to ask questions about the work conducted in the flight simulation lab. The result of the study will be made available to satisfy the curiosity of any participants on the matter.

D. POTENTIAL RISKS FROM THE STUDY

1. For each procedure used in this study, describe any known or foreseeable risks to the participants, including physiological, psychological, emotional, social and economic risks.

The use of motion platforms for flight simulation presents psychological and physiological risks.

Psychological risks include:

Disorientation and motion sickness.

Physiological risks include:

Injury could be caused by the unlikely event of a platform breakdown.

2. Describe the procedures or safeguards in place to protect the participants in light of the risks identified in D1.

Psychological risks:

Disorientation and motion sickness can occur when motion experienced by the passenger contradicts the visual cues he receives while flying the aircraft. This is unlikely to happen as the algorithm used to control the motion platform will produce motions that support the visual cues experienced in the aircraft. This has been experimentally verified using an inertial measurement unit used to measure motions experienced on the platform. The algorithms used to drive the motion platform are also well known and understood. In addition, a clearly identified pause button is available to the passenger on the flight simulator. He can press this button at any time to interrupt the simulation and center the motion platform. A verbal cue from the passenger can also alert the operator who can stop the simulation and center the motion platform. So far, no passenger of RMC's 3 DOF flight simulator has ever exhibited disorientation or motion sickness.

Physiological risks:

The system in use is controlled by a robust software interface which allows the operator to stop the motion platform at any time. This software was debugged and tested by other faculty members. Its behaviour is very stable and predictable. In addition, an emergency power switch (hardware) located close to the operator station can be used to completely stop the system at any time. This redundancy between software and hardware shutdown mechanism guarantees that the operator will always be able to stop the motion platform.

Although the motions generated by the system are not powerful enough to throw a passenger off the platform, the passenger is always secured to his seat with a four point harness bolted to the motion platform base. In the unlikely event of a platform breakdown, the passenger will remain securely fastened to his seat and the motion platform will be immediately de-energized.

Finally, a strict procedure as highlighted in Annex B is always be followed during the boarding and un-boarding of passengers. Overall, the operation of RMC's 3 DOF flight simulator is extremely safe.

E. INFORMED CONSENT PROCESS

1. a) If participants are to be given a Letter of Information and asked to sign a Written Consent, attach copies.

b) If free and informed consent is to be sought in some way other than in writing indicate how and justify.

☒ Information letter with verbal consent; attach a copy (Annex C)
☐ Information/cover letter; attach a copy
☐ Other "Specify" _____

Justification*:*

A cover letter will be included with the questionnaire which is reviewed by the participants prior to the conduct of any flight simulation. As the study does not collect any personal information on the participants a signature is not desired. Proceeding with the flight testing and answering the questionnaire indicates consent from the participants.

c) If consent will not be obtained from potential participants, provide a justification.

n/a

2. a) If this study involves persons who are not in a position to give their valid consent to participate, attach a copy of the Letter of Information and Permission Form to be used to obtain permission from those with legal authority to give it.

If voluntary assent at the time of participation is **not** a precondition for the incompetent individual's participation, provide a justification.

n/a

b) If this study involves persons for whom consent to participate in the research project is not appropriate in the context of their cultural background, indicate how their involvement will be negotiated, and how their consent will be obtained.

n/a

3. ***The Tri-Council Policy document suggests that community approval may be required when the research involves indigenous people, the cultural property of indigenous people or where individuals speak on behalf of an indigenous group.***

If this study requires the approval of an indigenous community as a whole, describe how this approval will be obtained.

n/a

F. PRIVACY OF PARTICIPANTS AND CONFIDENTIALITY OF DATA

1. Describe the procedures to be used to ensure privacy of participants and confidentiality of data both during the research and in the release of the findings.

The data will be obtained using the pilot rating questionnaire. The data collected is anonymous and does not identify the participants. The findings will only include pilot ratings and observations along with statistical information. The participants cannot be identified through this information.

2. How will written records, video/audio tapes, completed questionnaires and recordings be kept secure?

The questionnaires will be kept in a secure location during the research. After the research the questionnaires will be scanned and stored on a computer. The paper questionnaires will be destroyed after an electronic copy is available.

3. What is the final disposition of the data?

- ☐ Confidential shredding of the paper records
- ☐ Erasing of electronic data, audio/video tapes
- ☒ Data will be retained indefinitely in a secure location
- ☐ Other _____

4. If there are conditions under which privacy of participants or confidentiality of data cannot be guaranteed, provide details.

n/a

G. DECEPTION

1. If deception is to be used in this study, provide a justification.

n/a

2. If deception is to be used, outline the process to be used to disclose the full nature of the research to the participants. Attach a copy of the information sheet describing the true nature of the research.

n/a

Is consent sought again following disclosure of the full nature of the research to the participants for the use of their data**?** Yes ☐ No ☐

APPLICANT AGREEMENT

I have read the **Guiding Ethical Principles in the Tri-Council Policy Statement** and agree to comply with the conditions outlined in the **Guidelines**. In the case of student research, as Faculty Supervisor, my signature indicates that I have read and approved the application and proposal and I deem the project to be of high scientific, ethical and scholarly standards.

_____ _____

Signature of Applicant Date

_____ _____

Signature of Faculty Supervisor (if applicable) Date

INFORMED CONSENT LETTER

Good day,

The purpose of this study is to evaluate the fidelity of RMC's 3 DOF flight simulator. RMC's 3 DOF flight simulator is a low cost flight simulator when compared to other commercially available flight simulator. The study involves flying the simulator through two circuits (with and without motion). Then you will be asked to fill a short questionnaire to rate the flight simulator. It will take approximately 30 minutes of your time to complete these activities.

2Lt Jean Le Bouthillier of the Department of Mathematics and Computer Science is conducting the study.

Participation is voluntary; You are free to participate to the extent that you feel comfortable and can decide to stop participating in the study at any time by alerting the research staff.

Strict guidelines will be followed to protect your privacy. The ratings you provide about the flight simulator will be anonymous. Information on your flight experience will be collected to study the impact of flight experience on the results. It will not be possible to link the results with your participation in this study and only group data will be reported. 2Lt Jean Le Bouthillier will informally keep track of who has completed the study to ensure participants do not participate more than once.

This research project has received ethical approval by the Royal Military College Research Ethics Board. If you have concerns about the ethical nature of this study, please contact the Chair of this Committee: Dr. Daniel Lagace-Roy (Daniel.lagace-roy@rmc.ca, 613-541-6000 ext 6572).

Questions regarding this study should be addressed to 2Lt Jean Le Bouthillier, jean.le.bouthillier@rmc.ca, tel: 613 583-8778. The results of this study (when available) can be made available upon request.

Proceeding with the flight testing and filling the rating questionnaire will be interpreted as consent to participate in this study.

If you wish to have a copy of this sheet, you may simply detach it from the questionnaire and take it away with you.Thank you for your time and consideration.

ETHICS APPROVAL LETTER

File number: 2010-22 MCS

Project title: The Design, Implementation and Study of a Low Cost 3 Degree-of-Freedom Flight Simulator using RMC'S 3 DOF Motion Platform

Principal investigator: 2Lt Jean Le Bouthillier

Co Investigator:

Supervisors: Dr. Yawei Liang

Date of submission: 15 October 2010

Anticipated commencement date: 25 October 2010

Anticipated completion date: 5 November 2010

Date of approval: 21 October 2010

Period of approval: 21 October 2010 to 20 October 2011

This is to inform you that RMC Research Ethics Board (REB) has granted approval to the above-mentioned project and it can now proceed. The approval is based only on the documents submitted and only in the language(s) presented. This approval is valid for 12 months. If the project goes beyond this date, you must inform REB and obtain approval for an extension.

Any intentional changes to the protocol, prior to the start of data collection must be submitted to and approved by the Chair.

Researchers should not proceed with a project if unforeseen changes to the protocol threaten participants' right to informed consent or place participants at a higher risk level than anticipated. Such unforeseen changes to the protocol during the conduct of the research must be communicated within four working days to the REB Chair, as well as the actions taken to protect the dignity of participants.

Any undesirable experience or response (adverse event) from participants during their involvement in the study must also be reported within four working days to the REB Chair, as well as actions taken by the research team to protect the participants. Such adverse event may be emotional, psychological, physiological, or physical in nature.

Signature: _____

Dr. J. Daniel Lagacé-Roy

Chair, RMC REB

Phone: 613-541-6000 x6572

E-mail: daniel.lagace-roy@rmc.ca

Annex K - C++ Implentation Code

The integrality of the C++ code used for the implementation of the flight simulation system is available in the pages to follow. The code has been printed with reduced spacing and without any empty lines for a more compact printing. Indeed, due to the length of some code lines it was not possible to print the code in the document without a substantial lost of space. The Galil.h and SimConnect.h code is not included as it is in the public domain and can be easily obtained from either Galil or Microsoft.

FSXConnect.h

```
/**
** FSXConnect.h
** IDE: Visual C++ 2008 Express Edition
** @author 2Lt Jean Le Bouthillier
** Latest Revision: 20 Sept 2010
**
*/
#pragma once
#ifndef FSX_CONNECT_H
#define FSX_CONNECT_H
#include &lt;winsock2.h&gt;
#include &lt;string&gt;
#include "SimConnect.h"  // Microsoft SimConnect C++ Library
#include "UdpSend.h"
using namespace std;
class FSXConnect{
   private:
   /**
   ** Flag used to notify the end of a connection
   */
   bool quit;
   /**
   ** Flag used to hold the connection status
   */
   bool connected;
   /**
   ** SimConnect handle to FSX
   */
   HANDLE hSimConnect;
   /**
   ** UDP Client to send data to Matlab
   */
   UdpSend udpSend;
   /**
   ** Number of UDP ports required
   */
   PortNumber udpDataPorts[8];
   /**
   ** Buffer for Aircraft Pitch and Bank values
   */
   double pitchBank[2];
   /**
   ** A structure for the inputs of FSX
   */
   struct Inputs {
      int id;  // An integer for the id of the simulation variable
      float value; // A float variable to hold the simulation variable
   };
```

```
   /**
   ** 6 maxReturnedItems as we request 3 accelerations and 3 angular velocities
   */
   #define maxReturnedItems 8
   /**
   ** A structure that is used to receive Tagged data
   */
   struct StructDatum {
      Inputs datum[maxReturnedItems];
   };// StructDatum
   /**
   ** An enumeration of the events we are interested in
   */
   static enum EVENT_PDR {
      EVENT_SIM_PAUSE,
   };// EVENT_PDR
   /**
   ** An enumeration of the definitions we are interested in
   */
   static enum DATA_DEFINE_ID {
      DEFINITION_PDR,
   };// DATA_DEFINE_ID
   /**
   ** An enumeration of the requests we are interested in
   */
   static enum DATA_REQUEST_ID {
      REQUEST_PDR,
   };// DATA_REQUEST_ID
   /**
   ** An enumeration of the simulation variables we are interested in
   */
   static enum DATA_NAMES {
      DATA_X_ACCELERATION,
      DATA_Y_ACCELERATION,
      DATA_Z_ACCELERATION,
      DATA_P_ANGULAR_VELOCITY,
      DATA_Q_ANGULAR_VELOCITY,
      DATA_R_ANGULAR_VELOCITY,
      DATA_P_AIRCRAFT_PITCH,
      DATA_B_AIRCRAFT_BANK
   };// DATA_NAMES
   /*
   ** This method must be static, since it is used as a callback.
   ** Will receive 'this' pointer in the pContext parameter.
   */
   static void CALLBACK MyDispatchProcPDR(SIMCONNECT_RECV* pData, DWORD cbData, void *pContext);
   public:
   FSXConnect(); // FSXConnect constructor
   void FSXConnectRequest(); // Method requesting a connection to FSX
   void FSXDisconnect();  // Method disconnecting from FSX
   void simulation(); // Method performing one simulation step
   bool isConnected(); // Method returning the connection status
};// FSXConnect
#endif // FSX_CONNECT_H
```

FSXConnect.cpp

```
/**
** FSXConnect.cpp
** IDE: Visual C++ 2008 Express Edition
** @author 2Lt Jean Le Bouthillier
** Latest Revision: 20 Sept 2010
**
** C++ SimConnect Data Request for fAA and wAA (inputs to the classical washout algorithm).
**
** fAA=[ax, ay, az]
** wAA=[p, q, r]
**
** Description:
```

```
** After a flight has loaded, the program will request the accelerations of the aircraft (referenced to its center of mass)
** and its angular velocities. Anytime the values change in the flight simulator, the program will return the accelerations in
** meters/s and the angular velocities in degrees/s. An FSX connect request is a subscription to some flight simulator variables.
** When the tagged variables change in the flight simulator, the program notifies a registered handle that is created in this class.
*/
#include "FSXConnect.h"
/***
** FSXConnect Constructor
*/
FSXConnect::FSXConnect(){
   quit = false;  // Set the end of connection flag to false
   connected = false; // There is no open connection at the instantiation
   hSimConnect = NULL; // Set the SimConnect handle to NULL for now
   /**
   ** The ports to be used for the UDP data transfer over to MATLAB
   ** NOTE: Ports below 1024 are priviledged and are not a good choice.
   */
   udpDataPorts[0] = 10101;
   udpDataPorts[1] = 10102;
   udpDataPorts[2] = 10103;
   udpDataPorts[3] = 10104;
   udpDataPorts[4] = 10105;
   udpDataPorts[5] = 10106;
   udpDataPorts[6] = 10110;
   udpDataPorts[7] = 10111;
   pitchBank[0] = 0;
   pitchBank[1] = 0;
   udpSend.Initialize("127.0.0.1"); // UDP protocol object instance
}// FSXConnect
/**
** The MyDispatchProcPDR method is where the data sent from the simulator is handled accordingly.
** This method is listening for events or data from FSX. In this case, it is listening for a
** pause or unpause event of the flight simulator in which case null data will be broadcasted.
** Next, it is listening for accelerations and angular velocity data from FSX. When it is
** determined that such data has been received from FSX, it is broadcasted on UDP on 6 different
** ports. By default, those ports are 10101 to 10106.
**
** @param pData SimConnect receive structure object
** @param cbData callback data allocator
** @param *pContext pointer to the context structure
*/
void CALLBACK FSXConnect::MyDispatchProcPDR(SIMCONNECT_RECV* pData, DWORD cbData, void *pContext){
   double zero = 0; // a value used to send zero data
   static bool paused; // flag used to keep the state of the pause/unpaused event from FSX
   /**
   ** Cast pContext to FSXConnect.
   ** This is the instance of FSXConnect that caused this callback to be called.
   ** 'thisFSXConnect' can be used like 'this' in any non-static method.
   */
   FSXConnect *thisFSXConnect = (FSXConnect *) pContext;
   switch(pData->dwID){ // Switch on SimConnect DWord ID
      /**
      ** If we receive some event from the simulator on the events we subscribed to, perform actions:
      */
      case SIMCONNECT_RECV_ID_EVENT:{
         SIMCONNECT_RECV_EVENT *evt = (SIMCONNECT_RECV_EVENT*)pData; // Cast the data into SimConnect object structure
         switch(evt->uEventID){ // Check the event ID
            case EVENT_SIM_PAUSE: // If it is the pause event
            if(evt->dwData == 1){ // Paused
               paused = true;
            }// if
            else if(evt->dwData == 0){ // Unpaused
               paused = false;
            }// else if
            break;
            default:
            break;
         }// switch
         break;
      }// case
```

```
/**
** If we receive some data from the simulator on the data we tagged, perform actions:
*/
case SIMCONNECT_RECV_ID_SIMOBJECT_DATA:{ // In case we receive a Simobject Data (simulation variable)
   SIMCONNECT_RECV_SIMOBJECT_DATA *pObjData = (SIMCONNECT_RECV_SIMOBJECT_DATA*)pData; // Cast pData into o
bject data
   switch(pObjData->dwRequestID){  // Obtain the id of the simObject
      case REQUEST_PDR:{
         int count = 0;
         StructDatum *pS = (StructDatum*)&pObjData->dwData;
         /**
         ** There can be a minimum of 1 and a maximum of maxReturnedItems
         ** in the StructDatum structure. The actual number returned will
         ** be held in the dwDefineCount parameter.
         */
         if(!paused){
            while (count &lt; (int) pObjData->dwDefineCount){ // For each element we will extract the data and send over UDP.
               /**
               ** Switch statement on the data received and the appropriate action.
               ** In this case an UDP broadcast on select port.
               */
               switch (pS->datum[count].id){
                  case DATA_X_ACCELERATION:
                  // Cast into double and send value over UDP
                  thisFSXConnect->udpSend.SendDouble(thisFSXConnect->udpDataPorts[0], (double)(pS->datum[count].value));
                  break;
                  case DATA_Y_ACCELERATION:
                  // Cast into double and send value over UDP
                  // Polarity is inverted for the y axis since FSX uses different convention than the SAE right hand rule
                  thisFSXConnect->udpSend.SendDouble(thisFSXConnect->udpDataPorts[1], -(double)(pS->datum[count].value));
                  break;
                  case DATA_Z_ACCELERATION:
                  // Cast into double and send value over UDP
                  thisFSXConnect->udpSend.SendDouble(thisFSXConnect->udpDataPorts[2], (double)(pS->datum[count].value));
                  break;
                  case DATA_P_ANGULAR_VELOCITY:
                  // Cast into double and send value over UDP
                  // Polarity is inverted for the roll since FSX uses different convention than the SAE right hand rule
                  thisFSXConnect->udpSend.SendDouble(thisFSXConnect->udpDataPorts[3], -(double)(pS->datum[count].value));
                  break;
                  case DATA_Q_ANGULAR_VELOCITY:
                  // Cast into double and send value over UDP
                  // Polarity is inverted for the pitch since FSX uses different convention than the SAE right hand rule
                  thisFSXConnect->udpSend.SendDouble(thisFSXConnect->udpDataPorts[4], -(double)(pS->datum[count].value));
                  break;
                  case DATA_R_ANGULAR_VELOCITY:
                  // Cast into double and send value over UDP
                  // Polarity is inverted for the pitch since FSX uses different convention than the SAE right hand rule
                  thisFSXConnect->udpSend.SendDouble(thisFSXConnect->udpDataPorts[5], -(double)(pS->datum[count].value));
                  break;
                  case DATA_P_AIRCRAFT_PITCH:
                  // Cast into double and send value over UDP
                  // Polarity is inverted for the pitch since FSX uses different convention than the SAE right hand rule
                  thisFSXConnect->udpSend.SendDouble(thisFSXConnect->udpDataPorts[6], -(double)(pS->datum[count].value));
                  break;
                  case DATA_B_AIRCRAFT_BANK:
                  // Cast into double and send value over UDP
                  // Polarity is inverted for the pitch since FSX uses different convention than the SAE right hand rule
                  thisFSXConnect->udpSend.SendDouble(thisFSXConnect->udpDataPorts[7], -(double)(pS->datum[count].value));
                  break;
                  default:
                  //printf("\nUnknown datum ID: %d", pS->datum[count].id); //To catch errors.
                  break;
               }// switch
               ++count;
            }// while
         }// if
         else if(paused){
            for(int i=0; i&lt;8; i++){
               thisFSXConnect->udpSend.SendDouble(thisFSXConnect->udpDataPorts[i], zero);
```

```
                }// for
            }// else if
            break;
        }// case
        default:
        break;
    }// switch
    break;
  }// case
  /**
  ** If the simulator sends the event "quit", we stop this program.
  */
  case SIMCONNECT_RECV_ID_QUIT:{
    thisFSXConnect->quit = true;
    break;
  }// case
  default: // This happens when the program has established a connection with the simulator but there is no flight underway.
  //printf("\nNo simulation running or Unknown Event ID: %d",pData->dwID);
  break;
 }// switch
}// MyDispatchProcPDR
/**
** In the FSXConnectRequest method a connection to SimConnect is opened
** and the variables we will be looking at are subscribed to.
** A simulation start event is also requested as it will be used
** to know when to start listening for the variables we are interested in.
*/
void FSXConnect::FSXConnectRequest(){
 HRESULT hr; // Handle return object
 /**
 ** If there was a successful connection to FSX
 */
 if (SUCCEEDED(SimConnect_Open(&hSimConnect, "Tagged Data", NULL, 0, 0, 0))){
  connected = true;  // Set the connection status flag to true
  /**
  ** Set up the data definition, ensuring that all the elements are in Float32 units, to
  ** match the StructDatum structure. The number of entries in the DEFINITION_PDR
  ** definition should be equal to the maxReturnedItems defined.
  ** There is an inversion in the axis to convert to the proper axis for the washout algorithm
  */
  hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_PDR, "ACCELERATION BODY Z", "meters per second squared",
  SIMCONNECT_DATATYPE_FLOAT32, 0, DATA_X_ACCELERATION);
  hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_PDR, "ACCELERATION BODY X", "meters per second squared",
  SIMCONNECT_DATATYPE_FLOAT32, 0, DATA_Y_ACCELERATION);
  hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_PDR, "ACCELERATION BODY Y", "meters per second squared",
  SIMCONNECT_DATATYPE_FLOAT32, 0, DATA_Z_ACCELERATION);
  hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_PDR, "ROTATION VELOCITY BODY Z", "degrees per second",
  SIMCONNECT_DATATYPE_FLOAT32, 0, DATA_P_ANGULAR_VELOCITY);
  hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_PDR, "ROTATION VELOCITY BODY X", "degrees per second",
  SIMCONNECT_DATATYPE_FLOAT32, 0, DATA_Q_ANGULAR_VELOCITY);
  hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_PDR, "ROTATION VELOCITY BODY Y", "degrees per second",
  SIMCONNECT_DATATYPE_FLOAT32, 0, DATA_R_ANGULAR_VELOCITY);
  hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_PDR, "PLANE PITCH DEGREES", "degrees",
  SIMCONNECT_DATATYPE_FLOAT32, 0, DATA_P_AIRCRAFT_PITCH);
  hr = SimConnect_AddToDataDefinition(hSimConnect, DEFINITION_PDR, "PLANE BANK DEGREES", "degrees",
  SIMCONNECT_DATATYPE_FLOAT32, 0, DATA_B_AIRCRAFT_BANK);
  /**
  ** Request a subscribtion to a simulation start event
  */
  hr = SimConnect_SubscribeToSystemEvent(hSimConnect, EVENT_SIM_PAUSE, "Pause");
  hr = SimConnect_RequestDataOnSimObject(hSimConnect, REQUEST_PDR, DEFINITION_PDR,
  SIMCONNECT_OBJECT_ID_USER, SIMCONNECT_PERIOD_VISUAL_FRAME,
  SIMCONNECT_DATA_REQUEST_FLAG_CHANGED | SIMCONNECT_DATA_REQUEST_FLAG_TAGGED );
 }// if
}// FSXConnectRequest
/**
** In the FSXDisconnect method, the Simconnect
** connection is terminated.
*/
void FSXConnect::FSXDisconnect(){
```

```
   HRESULT hr; // Handle return object
   hr = SimConnect_Close(hSimConnect); // close the connection to SimConnect
   connected = false; // Set the connection status flag to false
}// FSXDisconnectRequest
/**
** In the simulation method a call is made to SimConnect.
** It is important to add 'this' as the last parameter
** in order to receive it the callback. This method returns
** the state of the connection.
*/
void FSXConnect::simulation(){
   if(!quit){
      SimConnect_CallDispatch(hSimConnect, MyDispatchProcPDR, this); // Simulation step
   }// if
   else if (quit){
      connected = false;
   }// else
}// simulation
/**
** The isConnected method returns the status
** of the FSX connection.
**
** @return &lt;true&gt; if FSXConnect is connected to FSX, &lt;false&gt; if there is no connection
*/
bool FSXConnect::isConnected(){
   return connected;
}// isConnected
```

Mainwindow.h

```
/**
** Mainwindow.h
** IDE: Visual C++ 2008 Express Edition
** @author 2Lt Jean Le Bouthillier
** Latest Revision: 20 Sept 2010
**
*/
#pragma once
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include "Galil.h"
#include "Mediator.h"
#include &lt;QtGui/QMainWindow&gt;
#include &lt;Windows.h&gt;
#include &lt;cstddef&gt;
#include &lt;string&gt;
#include &lt;cassert&gt;
#include &lt;iostream&gt;
using namespace std;
namespace Ui{
   class MainWindow;
}// Ui
class MainWindow : public QMainWindow{
   Q_OBJECT
   public:
   MainWindow(QWidget *parent = 0);
   ~MainWindow();
   private:
   /**
   ** Flag for the feedback option
   */
   bool feedback;
   /**
   ** Flag for the motion simulation
   */
   bool motionSimulation;
   /**
   ** String to hold connection information
   */
```

```
string platformType;
/**
** Pointer to a mediator object
*/
Mediator *m;
/**
** Pointer to a ui object
*/
Ui::MainWindow *ui;
string Q2S(QString qs);
QString S2Q(string s);
void checkConnections();
void Feedback(string s);
void enableManualControls(bool );
private slots:
void on_FSXDisconnect_clicked();
void on_FSXConnect_clicked();
void on_extend_clicked();
void on_yawRight_released();
void on_yawRight_pressed();
void on_yawLeft_released();
void on_yawLeft_pressed();
void on_swayRight_released();
void on_swayRight_pressed();
void on_swayLeft_released();
void on_swayLeft_pressed();
void on_surgeBackward_released();
void on_surgeBackward_pressed();
void on_surgeForward_released();
void on_surgeForward_pressed();
void on_m6down_released();
void on_m6down_pressed();
void on_m6up_released();
void on_m6up_pressed();
void on_m5down_released();
void on_m5down_pressed();
void on_m5up_released();
void on_m5up_pressed();
void on_m4down_pressed();
void on_m4up_pressed();
void on_m4down_released();
void on_m4up_released();
void on_m3down_released();
void on_m3down_pressed();
void on_m3up_released();
void on_m3up_pressed();
void on_m2down_released();
void on_m2down_pressed();
void on_m2up_released();
void on_m2up_pressed();
void on_m1down_released();
void on_m1down_pressed();
void on_m1up_released();
void on_m1up_pressed();
void on_heaveDown_released();
void on_heaveDown_pressed();
void on_heaveUp_released();
void on_heaveUp_pressed();
void on_rollRight_released();
void on_rollRight_pressed();
void on_rollLeft_released();
void on_rollLeft_pressed();
void on_pitchBackward_released();
void on_pitchBackward_pressed();
void on_pitchForward_released();
void on_pitchForward_pressed();
void on_park_clicked();
void on_center_clicked();
void on_markPositionParked_clicked();
void on_erasePosition_clicked();
```

```
    void on_speedSlider_sliderMoved(int );
    void on_stop_button_clicked();
    void on_begin_button_clicked();
    void on_checkBox_stateChanged(int );
    void on_clearButton_clicked();
    void on_connectButton_clicked();
};// MainWindow
#endif // MAINWINDOW_H
```

Mainwindow.cpp

```
/**
** mainwindow.cpp
** IDE: Visual C++ 2008 Express Edition
** @author 2Lt Jean Le Bouthillier
** Latest Revision: 07 May 2010
**
** Description:
** This class contains the necessary code to implement the graphical user interface (GUI)
** that is used by the motion platform user. It includes GUI object listeners as well as
** some convenience methods to facilitate the operation of the GUI. When an event occurs
** on the GUI, the appropriate actions are taken and most of time the Mediator is made aware
** of the user input.
**
*/
#include "mainwindow.h"
#include "ui_mainwindow.h"
/**
** MainWindow Constructor
**
** @param *parent pointer to the QWidget parent
*/
MainWindow::MainWindow(QWidget *parent)
: QMainWindow(parent), ui(new Ui::MainWindow){
    feedback = true;  // Feedback on
    ui->setupUi(this);
    m = new Mediator();  // Initialize the Mediator
    motionSimulation = false;  // Set the motion simulation flag to false
}// MainWindow
/**
** MainWindow Destructor
*/
MainWindow::~MainWindow(){
    delete ui;
    m->execute("park");  // Send request to mediator
}// ~MainWindow
/**
** The Q2S method is used to change a QString to a std::string.
**
** @param qs the QString to be changed into a std::string
** @return the std::string obtained from the QString
*/
string MainWindow::Q2S(QString qs){
    return string((const char *)qs.toLocal8Bit());
}// Q2S
/**
** The S2Q method is used to change a std::string to a QString.
**
** @param s the String to be changed into a QString
** @return the Qstring obtained from the std::string
*/
QString MainWindow::S2Q(string s){
    return QString::fromLocal8Bit(s.c_str());
}// S2Q
/**
** Event handler for a click on the "Connect Button"
** This method asks the mediator toattempts a connection
** to a controller at the IP adress specified by the user
*/
```

```
void MainWindow::on_connectButton_clicked(){
   //ui->feedback->clear();  // Clear the display
   Feedback("Attempting to connect...");  // Feedback message
   platformType = m->connect(Q2S(ui->ipAddress->text()));  // Ask the mediator to attempt a connection at the specified IP
   if(platformType != "No Controllers Detected!"){ // Checking to see if a connection was established
      ui->connectedTo->setText("Connected to: " + S2Q(platformType)); // Updating the GUI with either "3DOF" or "6DOF"
      Feedback(m->execute("markPositionParked"));  // Ask the mediator to mark the initial platform position as parked
      Feedback(m->execute("center")); // Center the motion platform
      ui->connectButton->setDisabled(true);  // Disable the "Connect" button
      ui->FSXConnect->setEnabled(true); // Enable the "Begin Motion Simulation" button
      enableManualControls(true);  // Enable the manual controls
   }// if
}// on_connectButton_clicked
/**
** This method is used to check the status of the
** UDP and FSX connections. Once it has established
** the status of those connections, it enables or
** disables the appropriate buttons on the GUI.
*/
void MainWindow::checkConnections(){
   while(1){  // When there is no motion simulation
      if(m->isReceivingUDP() && m->isFSXConnected()){ // If both FSX and Simulink connection are open
         ui->SimulinkConnectLabel->setText("Simulink Connection: Established!");
         ui->FSXConnectLabel->setText("FSX Connection: Established!");
         if(motionSimulation){
            ui->stop_button->setEnabled(true); // Enabled the "Stop Motion Simulation" button
            ui->begin_button->setDisabled(true); // Disable the "Begin Motion Simulation" button
            ui->FSXConnect->setDisabled(true); // Disable the "Start FSX connection" button
            ui->FSXDisconnect->setDisabled(true); // Disable the "Stop FSX conenction" button
         }// if
         else if(!motionSimulation){
            ui->begin_button->setEnabled(true); // Enable the "Begin Motion Simulation" button
            ui->stop_button->setDisabled(true); // Disable the "Stop Motion Simulation" button
            ui->FSXConnect->setDisabled(true); // Disable the "Start FSX connection" button
            ui->FSXDisconnect->setEnabled(true); // Enable the "Stop FSX conenction" button
         }// else if
         break;
      }// if
      else if (!m->isReceivingUDP() && !m->isFSXConnected()){
         ui->begin_button->setDisabled(true); // Disable the "Begin Motion Simulation" button
         ui->stop_button->setDisabled(true);  // Disable the "Stop Motion Simulation" button
         ui->FSXConnect->setEnabled(true);  // Enable the "Start FSX connection" button
         ui->FSXDisconnect->setDisabled(true); // Disable the "Stop FSX conenction" button
         ui->SimulinkConnectLabel->setText("Simulink Connection: Disconnected!");
         ui->FSXConnectLabel->setText("FSX Connection: Disconnected!");
      }//else if
      else if (m->isReceivingUDP() && !m->isFSXConnected()){
         ui->begin_button->setDisabled(true); // Disable the "Begin Motion Simulation" button
         ui->stop_button->setDisabled(true);  // Disable the "Stop Motion Simulation" button
         ui->FSXConnect->setEnabled(true);  // Enable the "Start FSX connection" button
         ui->FSXDisconnect->setDisabled(true); // Disable the "Stop FSX conenction" button
         ui->SimulinkConnectLabel->setText("Simulink Connection: Established!");
         ui->FSXConnectLabel->setText("FSX Connection: Disconnected!");
      }//else if
      else if (!m->isReceivingUDP() && m->isFSXConnected()){
         ui->begin_button->setDisabled(true); // Disable the "Begin Motion Simulation" button
         ui->stop_button->setDisabled(true);  // Disable the "Stop Motion Simulation" button
         ui->FSXConnect->setDisabled(true);  // Enable the "Start FSX connection" button
         ui->FSXDisconnect->setEnabled(true); // Disable the "Stop FSX conenction" button
         ui->SimulinkConnectLabel->setText("Simulink Connection: Disconnected!");
         ui->FSXConnectLabel->setText("FSX Connection: Established!");
      }//else if
      QApplication::processEvents(); // Allow the GUI to process events
   }//while
}// checkConnections
/**
** The Feedback method is used to quickly append a text
** to the user feedback window. The method accepts an
** std::string and performs the change to QString.
**
```

```
** @param s the text to be appended to the user feedback window
*/
void MainWindow::Feedback(string s){
   if(feedback == true)  // Check if the feedback option is enabled
   ui->feedback->appendPlainText(S2Q(s+"\n")); // Carriage return and append new text
}// Feedback
/**
** Event handler for the "Start FSX Connection" button.
** The mediator attempts a connection to FSX when the user
** clicks the button. The connection request is passed to
** the mediator in this method.
*/
void MainWindow::on_FSXConnect_clicked(){
   m->execute("FSXConnect");
   Feedback("Attempting connection to FSX...");
   checkConnections();  // Check connections
}// on_FSXConnect_clicked
/**
** Event handler for the "Stop FSX Connection" button.
** The mediator disconnects from FSX when the user
** presses the button. The disconnect request is passed
** to the mediator in this method.
*/
void MainWindow::on_FSXDisconnect_clicked(){
   m->execute("FSXDisconnect");
   Feedback("Disconnecting from FSX..."); // Feedback message
   checkConnections(); // Check connections
}// on_FSXDisconnect_clicked
/**
** Event handler for the "Begin Motion Simulation" button.
** When the user presses this button, the motion simulation begins.
** As such, the mediator is ask the begin the motion simulation.
*/
void MainWindow::on_begin_button_clicked(){
   motionSimulation = true;  // Set the motion simulation flag
   enableManualControls(false); // Disable manual controls of the platform
   Feedback("Motion simulation in progress..."); // Feedback message
   checkConnections(); // Check connections
   while(motionSimulation == true){
      m->simulation();
      QApplication::processEvents(); // Allow the GUI to process events
   }// while
}// on_begin_button_clicked
/**
** Event handler for the "Stop Motion Simulation" button.
** When this button is pressed by the user, the platform
** is parked and the program closes.
*/
void MainWindow::on_stop_button_clicked(){
   motionSimulation = false; // Stop the motion simulation
   Feedback("Motion simulation terminating..."); // Feedback message
   Feedback("Please wait a moment while the last motion simulation commands are executed..."); // Feedback message
   while(m->execute("stopMovement") == "Cannot execute command!"){ // Center the motion platform
      QApplication::processEvents();
   }// while
   Feedback(m->execute("center"));
   ui->stop_button->setDisabled(true); // Disable the "Stop Motion Simulation" button
   enableManualControls(true);  // Enable the manual controls of the platform
   checkConnections(); // Check connections
   //MainWindow::~MainWindow(); // Close the GUI, terminate the execution
   //exit(1);  // Exit the program
}// on_stop_button_clicked
/**
** Event handler for a click on the "Clear Display" button.
** The feedback window is cleared.
*/
void MainWindow::on_clearButton_clicked(){
   ui->feedback->clear(); // Clear the feedback window
}// on_clearButton_clicked
/**
```

```
** Event handler for a check in the "Feedback ON/OFF"
** checkbox. In this method, the feedback window is
** enabled or disabled according to the user input.
**
** @param s the state of the checkbox (0 is unchecked, 2 is checked)
*/
void MainWindow::on_checkBox_stateChanged(int s){
   if(s==0){ // If the checkbox is unchecked
      ui->feedback->setDisabled(true); // Disable the feedback text area
      feedback = false; // Set the feedback flag to false
   }// if
   else if(s==2){
      ui->feedback->setEnabled(true); // Enable the feedback text area
      feedback = true; // Set the feedback flag to true
   }// else
} // on_checkBox_stateChanged
/**
** Event handler for the speed slider.
** When the slider is moved, the speed of the motion
** platform must be changed. Everytime the value changes
** it is sent to the mediator.
*/
void MainWindow::on_speedSlider_sliderMoved(int ){
   int slidervalue = ui->speedSlider->value();
   ui->speedLabel->setText("Speed: " + QString::number(slidervalue, 10) + " %");
   m->change_speed(slidervalue);
}// on_speedSlider_sliderMoved
/**
** Event handler for the "Erase Parked Position" button.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_erasePosition_clicked(){
   string status = m->execute("erasePosition"); // Send request to mediator
   if(status != "Cannot execute command!"){ // Check if the command can be executed
      Feedback(status);  // Let the user know the status
      ui->erasePosition->setDisabled(true); // Disable the "Erase Parked Position" button
      ui->markPositionParked->setEnabled(true); // Enable the "Mark Position as Parked" button
   }// if
   else{  // If the command cannot be executed
      Feedback(status);  // Let the user know the status
   }// else
}// on_erasePosition_clicked
/**
** Event handler for the "Mark Position as Parked" button.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_markPositionParked_clicked(){
   string status = m->execute("markPositionParked");  // Send request to mediator
   if(status != "Cannot execute command!"){ // Check if the command can be executed
      Feedback(status);  // Let the user know the status
      ui->markPositionParked->setDisabled(true); // Disable the "Mark Position as Parked" button
      ui->erasePosition->setEnabled(true); // Enable the "Erase Parked Position" button
   }// if
   else{  // If the command cannot be executed
      Feedback(status);  // Let the user know the status
   }// else
}// on_markPositionParked_clicked
/**
** Event handler for the "Center" button.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_extend_clicked(){
   Feedback(m->execute("extend")); // Send request to mediator
}// on_Extend_clicked
/**
** Event handler for the "Extend" button.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_center_clicked(){
   Feedback(m->execute("center")); // Send request to mediator
```

```
}// on_center_clicked
/**
** Event handler for the "Park" button.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_park_clicked(){
    Feedback(m->execute("park")); // Send request to mediator
}// on_park_clicked
/**
** This method enables of disables the manual motion
** platform controls as warranted by the state of execution
** of the program-> (i.e. the manual controls are disabled
** during flight simulation)
*/
void MainWindow::enableManualControls(bool b){
    if(b == true){ // If asked to enable manual controls
        ui->speedSlider->setEnabled(true);
        ui->pitchForward->setEnabled(true);
        ui->pitchBackward->setEnabled(true);
        ui->rollLeft->setEnabled(true);
        ui->rollRight->setEnabled(true);
        ui->heaveUp->setEnabled(true);
        ui->heaveDown->setEnabled(true);
        ui->m1up->setEnabled(true);
        ui->m1down->setEnabled(true);
        ui->m2up->setEnabled(true);
        ui->m2down->setEnabled(true);
        ui->m3up->setEnabled(true);
        ui->m3down->setEnabled(true);
        ui->extend->setEnabled(true);
        ui->center->setEnabled(true);
        ui->park->setEnabled(true);
        ui->erasePosition->setEnabled(true);
        ui->center->setEnabled(true);
        if(platformType=="6DOF"){ // If connected to the 6DOF: Additional options
            ui->m4up->setEnabled(true);
            ui->m4down->setEnabled(true);
            ui->m5up->setEnabled(true);
            ui->m5down->setEnabled(true);
            ui->m6up->setEnabled(true);
            ui->m6down->setEnabled(true);
            ui->swayLeft->setEnabled(true);
            ui->swayRight->setEnabled(true);
            ui->surgeForward->setEnabled(true);
            ui->surgeBackward->setEnabled(true);
            ui->rollLeft->setEnabled(true);
            ui->rollRight->setEnabled(true);
        }// if
    }// if
    else if(b == false){  // if asked to disable manual controls
        ui->speedSlider->setDisabled(true);
        ui->pitchForward->setDisabled(true);
        ui->pitchBackward->setDisabled(true);
        ui->rollLeft->setDisabled(true);
        ui->rollRight->setDisabled(true);
        ui->heaveUp->setDisabled(true);
        ui->heaveDown->setDisabled(true);
        ui->m1up->setDisabled(true);
        ui->m1down->setDisabled(true);
        ui->m2up->setDisabled(true);
        ui->m2down->setDisabled(true);
        ui->m3up->setDisabled(true);
        ui->m3down->setDisabled(true);
        ui->m4up->setDisabled(true);
        ui->m4down->setDisabled(true);
        ui->m5up->setDisabled(true);
        ui->m5down->setDisabled(true);
        ui->m6up->setDisabled(true);
        ui->m6down->setDisabled(true);
        ui->swayLeft->setDisabled(true);
```

```
        ui->swayRight->setDisabled(true);
        ui->surgeForward->setDisabled(true);
        ui->surgeBackward->setDisabled(true);
        ui->rollLeft->setDisabled(true);
        ui->rollRight->setDisabled(true);
        ui->extend->setDisabled(true);
        ui->center->setDisabled(true);
        ui->park->setDisabled(true);
        ui->erasePosition->setDisabled(true);
        ui->center->setDisabled(true);
        ui->markPositionParked->setDisabled(true);
    }// else if
}// enableManualControls
/**
** Event handler for the "Pitch Forward" button.
** As long as this button is pressed,
** the platform will pitch forward.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_pitchForward_pressed(){
    Feedback(m->execute("pitchForward"));  // Send request to mediator
}// on_pitchForward_pressed
/**
** Event handler for the "Pitch Forward" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_pitchForward_released(){
    Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_pitchForward_released
/**
** Event handler for the "Pitch Backward" button.
** As long as this button is pressed,
** the platform will pitch backward.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_pitchBackward_pressed(){
    Feedback(m->execute("pitchBackward"));  // Send request to mediator
}// on_pitchBackward_pressed
/**
** Event handler for the "Pitch Backward" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_pitchBackward_released(){
    Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_pitchBackward_released
/**
** Event handler for the "Roll Left" button.
** As long as this button is pressed,
** the platform will roll left.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_rollLeft_pressed(){
    Feedback(m->execute("rollLeft")); // Send request to mediator
}// on_rollLeft_pressed
/**
** Event handler for the "Roll Left" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_rollLeft_released(){
    Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on rollLeft_released
/**
** Event handler for the "Roll Right" button.
** As long as this button is pressed,
** the platform will roll left.
** The request is passed to the mediator in this method.
*/
```

```
void MainWindow::on_rollRight_pressed(){
   Feedback(m->execute("rollRight")); // Send request to mediator
}// on_rollRight_pressed
/**
** Event handler for the "Roll Right" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_rollRight_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_rollRight_released
/**
** Event handler for the "Heave up" button.
** As long as this button is pressed,
** the platform will heave up.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_heaveUp_pressed(){
   Feedback(m->execute("heaveUp")); // Send request to mediator
}// on_heaveUp_pressed
/**
** Event handler for the "Heave Up" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_heaveUp_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_heaveUp_released
/**
** Event handler for the "Heave down" button.
** As long as this button is pressed,
** the platform will heave down.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_heaveDown_pressed(){
   Feedback(m->execute("heaveDown")); // Send request to mediator
}// on_heaveDown_pressed
/**
** Event handler for the "Heave Down" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_heaveDown_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_heaveDown_released
/**
** Event handler for the "Surge forward" button.
** As long as this button is pressed,
** the platform will surge forward.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_surgeForward_pressed(){
   Feedback(m->execute("surgeForward")); // Send request to mediator
}// on_surgeForward_pressed
/**
** Event handler for the "Surge forward" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_surgeForward_released(){
   Feedback(m->execute("stopMovement")); // Send request to mediator
}// on_surgeForward_released
/**
** Event handler for the "Surge backward" button.
** As long as this button is pressed,
** the platform will surge backward.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_surgeBackward_pressed(){
   Feedback(m->execute("surgeBackward")); // Send request to mediator
```

```
}// on_surgeBackward_pressed
/**
** Event handler for the "Surge backward" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_surgeBackward_released(){
   Feedback(m->execute("stopMovement")); // Send request to mediator
}// on_surgeBackward_released
/**
** Event handler for the "Sway left" button.
** As long as this button is pressed,
** the platform will sway left.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_swayLeft_pressed(){
   Feedback(m->execute("swayLeft")); // Send request to mediator
}// on_swayLeft_pressed
/**
** Event handler for the "Sway left" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_swayLeft_released(){
   Feedback(m->execute("stopMovement")); // Send request to mediator
}// on_swayLeft_released
/**
** Event handler for the "Sway right" button.
** As long as this button is pressed,
** the platform will sway right.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_swayRight_pressed(){
   Feedback(m->execute("swayRight")); // Send request to mediator
}// on_swayRight_pressed
/**
** Event handler for the "Sway right" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_swayRight_released(){
   Feedback(m->execute("stopMovement")); // Send request to mediator
}// on_swayRight_released
/**
** Event handler for the "Yaw left" button.
** As long as this button is pressed,
** the platform will yaw left.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_yawLeft_pressed(){
   Feedback(m->execute("yawLeft")); // Send request to mediator
}// on_yawLeft_pressed
/**
** Event handler for the "Yaw left" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_yawLeft_released(){
   Feedback(m->execute("stopMovement")); // Send request to mediator
}// on_yawLeft_released
/**
** Event handler for the "Yaw right" button.
** As long as this button is pressed,
** the platform will yaw right.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_yawRight_pressed(){
   Feedback(m->execute("yawRight")); // Send request to mediator
}// on_yawRight_pressed
/**
```

```
** Event handler for the "Yaw right" button.
** When the button is released, the motion stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_yawRight_released(){
   Feedback(m->execute("stopMovement")); // Send request to mediator
}// on_yawRight_released
/**
** Event handler for the "Motor 1 Up" button.
** As long as this button is pressed,
** actuator 1 will move up.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m1up_pressed(){
   Feedback(m->execute("actuator1Up"));  // Send request to mediator
}// on_m1up_pressed
/**
** Event handler for the "Motor 1 Up" button.
** When the button is released, the actuator stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m1up_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m1up_released
/**
** Event handler for the "Motor 1 Down" button.
** As long as this button is pressed,
** actuator 1 will move down.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m1down_pressed(){
   Feedback(m->execute("actuator1Down"));  // Send request to mediator
}// on_m1down_pressed
/**
** Event handler for the "Motor 1 Down" button.
** When the button is released, the actuator stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m1down_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m1down_released
/**
** Event handler for the "Motor 2 Up" button.
** As long as this button is pressed,
** actuator 2 will move up.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m2up_pressed(){
   Feedback(m->execute("actuator2Up"));  // Send request to mediator
}// on_m2up_pressed
/**
** Event handler for the "Motor 2 Up" button.
** When the button is released, the actuator stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m2up_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m2up_released
/**
** Event handler for the "Motor 2 Down" button.
** As long as this button is pressed,
** actuator 2 will move down.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m2down_pressed(){
   Feedback(m->execute("actuator2Down"));  // Send request to mediator
}// on_m2down_pressed
/**
** Event handler for the "Motor 2 Down" button.
** When the button is released, the actuator stops.
```

```
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m2down_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m2down_released
/**
** Event handler for the "Motor 3 Up" button.
** As long as this button is pressed,
** actuator 3 will move up.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m3up_pressed(){
   Feedback(m->execute("actuator3Up"));  // Send request to mediator
}// on_m3up_pressed
/**
** Event handler for the "Motor 3 Up" button.
** When the button is released, the actuator stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m3up_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m3up_released
/**
** Event handler for the "Motor 3 Down" button.
** As long as this button is pressed,
** actuator 3 will move down.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m3down_pressed(){
   Feedback(m->execute("actuator3Down"));  // Send request to mediator
}// on_m3down_pressed
/**
** Event handler for the "Motor 3 Down" button.
** When the button is released, the actuator stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m3down_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m3down_released
/**
** Event handler for the "Motor 4 Up" button.
** As long as this button is pressed,
** actuator 4 will move up.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m4up_pressed(){
   Feedback(m->execute("actuator4Up"));  // Send request to mediator
}// on_m4up_pressed
/**
** Event handler for the "Motor 4 Up" button.
** When the button is released, the actuator stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m4up_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m4up_released
/**
** Event handler for the "Motor 4 Down" button.
** As long as this button is pressed,
** actuator 4 will move down.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m4down_pressed(){
   Feedback(m->execute("actuator4Down"));  // Send request to mediator
}// on_m4down_pressed
/**
** Event handler for the "Motor 4 Up" button.
** When the button is released, the actuator stops.
** The request is passed to the mediator in this method.
*/
```

```
void MainWindow::on_m4down_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m4down_released
/**
** Event handler for the "Motor 5 Up" button.
** As long as this button is pressed,
** actuator 5 will move up.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m5up_pressed(){
   Feedback(m->execute("actuator5Up"));  // Send request to mediator
}// on_m5up_pressed
/**
** Event handler for the "Motor 5 Up" button.
** When the button is released, the actuator stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m5up_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m5up_released
/**
** Event handler for the "Motor 5 Down" button.
** As long as this button is pressed,
** actuator 5 will move down.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m5down_pressed(){
   Feedback(m->execute("actuator5Down"));  // Send request to mediator
}// on_m5down_pressed
/**
** Event handler for the "Motor 5 Down" button.
** When the button is released, the actuator stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m5down_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m5down_released
/**
** Event handler for the "Motor 6 Up" button.
** As long as this button is pressed,
** actuator 6 will move up.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m6up_pressed(){
   Feedback(m->execute("actuator6Up"));  // Send request to mediator
}// on_m6up_pressed
/**
** Event handler for the "Motor 6 Up" button.
** When the button is released, the actuator stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m6up_released(){
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m6up_released
/**
** Event handler for the "Motor 6 Down" button.
** As long as this button is pressed,
** actuator 6 will move down.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m6down_pressed(){
   Feedback(m->execute("actuator6Down"));  // Send request to mediator
}// on_m6down_pressed
/**
** Event handler for the "Motor 6 Down" button.
** When the button is released, the actuator stops.
** The request is passed to the mediator in this method.
*/
void MainWindow::on_m6down_released(){
```

```
   Feedback(m->execute("stopMovement"));  // Send request to mediator
}// on_m6down_released
```

Mediator.h

```
/**
** Mediator.h
** IDE: Visual C++ 2008 Express Edition
** @author 2Lt Jean Le Bouthillier
** Latest Revision: 11 May 2010
**
*/
#pragma once
#ifndef MEDIATOR_H
#define MEDIATOR_H
#include "FSXConnect.h"
#include "UdpReceive.h"
#include "Model.h"
#include &lt;string&gt;
#include &lt;math.h&gt;
#include &lt;iostream&gt;
#include &lt;sstream&gt;
using namespace std;
class Mediator{
   public:
   Mediator();
   string execute(string uInput);
   void simulation();
   string connect(string ipAddress);
   bool isReceivingUDP();
   bool isFSXConnected();
   void change_speed(int s);
   private:
   /**
   ** Pointer to a Model object
   */
   Model *m;
   /**
   ** Pointer to an FSXConnect object
   */
   FSXConnect *f;
   /**
   ** Double dynamic array for receiving the actuator lengths
   */
   double *lengths;
   /**
   ** Double dynamic array for receiving the actuator velocities
   */
   double *velocities;
   /**
   ** Double dynamic array for receiving the actuator accelerations
   */
   double *accelerations;
   /**
   ** UDP server object for lenghts
   */
   UdpReceive udpLengths;
   /**
   ** UDP server object for velocities
   */
   UdpReceive udpVelocities;
   /**
   ** UDP server object for accelerations
   */
   UdpReceive udpAccelerations;
   /**
```

```
   ** Flag to differentiate between 3 and 6DOF
   */
   string platformType;
   /**
   ** Value containing the number of actuators
   */
   int number_of_actuator;
};
#endif // MEDIATOR_H
```

Mediator.cpp

```
/**
** Mediator.cpp
** IDE: Visual C++ 2008 Express Edition
** @author 2Lt Jean Le Bouthillier
** Latest Revision: 11 May 2010
**
** Description:
** This class is responsible for providing a centralized location where interactions between all the
** object of the system are controlled. As per the mediator pattern, this class effectively
** encapsulate how all the object of the system interact. Coupling is reduced by this approach as
** it prevents objects from explicitely calling or reffering to each other, they must go through the
** mediator. The advantage of de-coupling is that the modification of one of the components of the
** system has little impact on the other components. More specifically, the mediator abstracts the
** motion platform from the GUI, the motion cueing algorithm and the Flight simulation environment.
** In other words, the GUI nor the algorithm or flight simulation environment knows anything about
** the motion platform.
**
*/
#include "Mediator.h"
/**
** Mediator Constructor
*/
Mediator::Mediator(){
   udpLengths.Initialize("127.0.0.1", 10107);  // Initialize the lengths UDP server
   udpVelocities.Initialize("127.0.0.1", 10108); // Initialize the velocities UDP server
   udpAccelerations.Initialize("127.0.0.1", 10109); // Initialize the accelerations UDP server
   m = new Model(); // Initialize the Model
   f = new FSXConnect(); // Creates an FSXConnect object
}// Mediator
/**
** The connect method is called by the GUI to perform a connection.
** This method simply passes the request to the model
** which will attempt a connection and return wether the connnection
** was successful (3DOF or 6DOF) or unsuccessful.
**
** @param ipAddress a string containing the ip adress of the controller
** @return a string containing the platform type either "3DOF" or "6DOF"
*/
string Mediator::connect(string ipAddress){
   platformType = m->connect(ipAddress);  // Pass the connection request to the model
   if(platformType == "3DOF"){
      number_of_actuator = 3;
      lengths = new double[number_of_actuator]; // Setting the size of the double array for lengths
      velocities = new double[number_of_actuator];  // Setting the size of the double array for velocities
      accelerations = new double[number_of_actuator];  // Setting the size of the double array for accelerations
   }// if
   else if(platformType == "6DOF"){
      number_of_actuator = 6;
      lengths = new double[number_of_actuator]; // Setting the size of the double array for lengths
      velocities = new double[number_of_actuator];  // Setting the size of the double array for velocities
      accelerations = new double[number_of_actuator];  // Setting the size of the double array for accelerations
   }// else if
   return platformType;
}// connect
/**
** The execute method is called to perform an action.
** This method simply passes the request to the model
```

```
** which will evaluate it and send the proper motion commands
** to the motion platform.
**
** @param uInput a string containing the command request
** @return  a string containing feedback on the success or failure of the request
*/
string Mediator::execute(string uInput){
   string status = "";
   if(uInput == "FSXConnect"){ // If the GUI is asking for an FSX connection
      f->FSXConnectRequest(); // Pass the FSX connect request to FSXConnect
   }// if
   else if (uInput == "FSXDisconnect"){ // If the GUI is asking for an FSX disconnection
      f->FSXDisconnect();  // Pass the FSX disconnect to FSXConnect
   }// else if
   else{  // For any other GUI requests
      status = m->process_user_input(uInput); // Pass the command request to the model
   }// else
   return status;
}// process_user_input
/**
** The change_speed method is called to change the motion platform speed.
** This method simply passes the information to the model
** which will interpret it properly and change the motion platform speed.
**
** @param s the new speed of the motion platform (from 1% to 100%)
*/
void Mediator::change_speed(int s){
   m->change_speed(s); // Pass the speed change request to the model
}// change_speed
/**
** The isReceivingUDP method checks if the mediator
** is receiving any UDP packets from MATLAB.
**
** @return &lt;true&gt; if UDP packets are received, &lt;false&gt; if no UDP packet are received
*/
bool Mediator::isReceivingUDP(){
   if(udpVelocities.ReceiveDoubleArray(velocities, number_of_actuator, 10) == true
   || udpAccelerations.ReceiveDoubleArray(accelerations, number_of_actuator, 10) == true
   || udpLengths.ReceiveDoubleArray(lengths, number_of_actuator, 10) == true){
      return true; // If there are any packets received, return true.
   }// if
   else{
      return false; // No packets were received, return false.
   }// else
}// isReceivingUDP
/**
** The isReceivingUDP method checks if FSXConnect is
** connected to FSX through SimConnect.
**
** @return &lt;true&gt; if there is an active FSX connection, &lt;false&gt; if there is no connection
*/
bool Mediator::isFSXConnected(){
   return f->isConnected();
}// isFSXConnected
/**
** The simulation method receives the motion simulation
** information by UDP from Simulink and forwards them to
** the model for processing.
*/
void Mediator::simulation(){
   f->simulation(); // Pass the simulation request to FSXConnect
   // Receive the data by UDP and pass it to the model.
   if (udpVelocities.ReceiveDoubleArray(velocities, number_of_actuator, 1) == true){  // Setting -
1 for the timeout acts as a spinlock (will wait undefinitely)
      m->velocity_command(velocities); // Sending the velocities to the model
   }// if
   if (udpAccelerations.ReceiveDoubleArray(accelerations, number_of_actuator, 1) == true){ // Setting -
1 for the timeout acts as a spinlock (will wait undefinitely)
      m->acceleration_command(accelerations);  // Sending the accelerations to the model
   }// if
```

```
   if (udpLengths.ReceiveDoubleArray(lengths, number_of_actuator, 1) == true){ // Setting -
1 for the timeout acts as a spinlock (will wait undefinitely)
      m->length_command(lengths);  // Sending the lengths to the model
   }// if
}// MotionSimulation
```

Model.h

```
/**
** Model.h
** IDE: Visual C++ 2008 Express Edition
** @author 2Lt Jean Le Bouthillier
** Latest Revision: 11 May 2010
**
*/
#pragma once
#ifndef MODEL_H
#define MODEL_H
#include "Galil.h"
#include &lt;string&gt;
#include &lt;iostream&gt;
#include &lt;sstream&gt;
#include &lt;math.h&gt;
using namespace std;
class Model{
   public:
   void change_speed(int s);
   string connect(string ip);
   string process_user_input(string uInput);
   void length_command(double *command);
   void velocity_command(double *command);
   void acceleration_command(double *command);
   bool is_platform_moving();
   private:
   /**
   ** Pointer to a Galil object for RMC's 3DOF
   */
   Galil *g;
   /**
   ** String used to differentiate between 3 and 6DOF
   */
   string platformType;
   /**
   ** The speed of the actuators for user inputs
   */
   string speed;
   /**
   ** String containing the controller ip adress
   */
   string ipAddress;
   /**
   ** String array for the lengths
   */
   string l[3];
   /**
   ** String array for the velocities
   */
   string v[3];
   /**
   ** String array for the accelerations
   */
   string a[3];
   /**
   ** Value containing the number of actuators
   */
   int number_of_actuator;
   /**
   ** Value containing the conversion factor from metric to 3/6DOF
   */
```

```
   int conversion_factor;
   /**
   ** Flag that is used to prevent any motion when set to &lt;true&gt;
   */
   bool prevent_movement;
   string int2string(int i);
   string make3DOFCommand(string prefix, double *v);
};
#endif // MODEL_H
```

Model.cpp

```
/**
** Model.cpp
** IDE: Visual C++ 2008 Express Edition
** @author 2Lt Jean Le Bouthillier
** Latest Revision: 11 May 2010
**
** For the Galil C++ API: http://www.galilmc.com/support/manuals/galiltools/library.html#libraryVersion
**
** Description:
** This class is responsible for interacting with RMC's 3DOF or 6DOF motion platforms.
** All motion platform specific operations are happening in this class. This model is
** able to detect which motion platform it is connected to and self-configure appropriately.
** Other classes can ask this model to process pre-programmed motions (i.e. park, m1 up, center)
** or they can ask the model to process length, velocity and acceleration commands. No platform
** information is required to communicate with the model. Because of this, this model can be
** used to interface RMC's motion platform with any other C++ application with ease.
**
*/
#include "Model.h"
/**
** Used to establish a connection with either RMC's 3DOF or
** 6DOF platform. Some parameters are also initialized depending
** on which platform the connection is established with.
**
** @param ip the ip adress of the controller to which a connection is attempted
** @return a string containing either "3DOF" or "6DOF"
*/
string Model::connect(string ip){
   ipAddress = ip;
   platformType = "3DOF"; // Default platform type (3DOF)
   try{ // Attempts to connect to the 3DOF
      g = new Galil(ipAddress); // Creates a Galil Object
      g->timeout_ms = 1000;  // Allow for a 1 second timeout
      g->connection(); // Connect to the controller
   }// try
   catch(string e){ // No 3DOF controller detected
      platformType = "6DOF";  // Change platform type for 6DOF
      try{  // Attempts to connect to the 6DOF
         /**
         **
         ** 6DOF CONNECTION CODE HERE
         **
         */
         platformType = "No Controllers Detected!"; // for now until there is a 6DOF
      }// try
      catch(...){  // (...) is a catch all form
         platformType = "No Controllers Detected!";
      }// catch
   }// catch
   if(platformType == "3DOF"){
      number_of_actuator = 3;  // Setting number of actuators
      conversion_factor = 30000;  // Metric to 3DOF conversion factor
      speed = "6750"; // Initialize the speed at 50% for 3DOF
      prevent_movement = false; // Initialize the prevent movement flag to false
   }// if
   else if(platformType == "6DOF"){
      number_of_actuator = 6;  // Setting number of actuators
```

```
      //conversion_factor = ?? // INSERT 6DOF CONVERSION FACTOR HERE
   }// if
   return platformType; // The connection information is returned
}// connect
/**
** Used to change the speed of the platform
** used for pre-programmed motions.
**
** @param s the new speed (a percentage from 1 to 100)
*/
void Model::change_speed(int s){
   if(platformType == "3DOF"){
      s *= 135;  // The maximal speed is 13500 (100%)
      speed = int2string(s);  // Set the motion platform speed string variable
   }// if
   else if (platformType == "6DOF"){
      // 6DOF CODE HERE
      speed = int2string(s);
   }// else if
}// change_speed
/**
** Used to transform an integer into a string.
**
** @param i the integer to be transformed into a string
** @return the string version of the integer
*/
string Model::int2string(int i){
   ostringstream ss;
   ss.str("");
   ss &lt;&lt; i;
   return ss.str();
}// int2string
/**
** Takes a prefix (i.e. PA, AC, SP ...) and an array of 3 doubles to
** generate a properly formatted 3DOF Galil command.
**
** @param prefix a string containing the command prefix
** @param *v a double array containing the three values of the command
** @return  a string containing the properly formatted 3DOF Galil command.
*/
string Model::make3DOFCommand(string prefix, double *v){
   ostringstream ss;
   ss.str("");
   ss.setf(ios::fixed,ios::floatfield);
   ss.precision(2);
   ss &lt;&lt; prefix &lt;&lt; " " &lt;&lt; v[0] &lt;&lt; "," &lt;&lt; v[1] &lt;&lt; "," &lt;&lt; v[2];
   return ss.str();
}// makeCommand
/**
** Used to send an actuator length command to
** the motion platform.
**
** @param *command a double array containing the three actuator lenghts
*/
void Model::length_command(double *command){
   for(int i=0; i&lt;number_of_actuator; i++){
      command[i] *= conversion_factor; // Converting metric to 3DOF units
      /**
      ** Here the 3DOF motion platform is limited in its lower reach to -5000 due to a design flaw of the platform.
      ** More information about this flaw and the limitation it forces upon the operation can be found in thesis.
      */
      if(command[i] &lt;= -5000)
      command[i] = -5000;
      if(command[i] &gt;= 6000)
      command[i] = 6000;
   }// for
   g->command("FL 6050,6050,6050"); // Define the forward limit slightly higher for flight simulation
   g->command("BL -5050,-5050,-5050"); // Define backward limit for all actuators (only 5050 because of platform design)
   g->command("PT1,1,1"); // Register all actuator for "on the fly" absolute position move
   g->command(make3DOFCommand("PA", command));  // Send the motion command to the controller
```

```
}// length_command
/**
** Used to send an actuator velocity command to
** the motion platform.
**
** @param *command a double array containing the three actuator velocity
*/
void Model::velocity_command(double *command){
   for(int i=0; i&lt;number_of_actuator; i++){
      command[i] *= conversion_factor; // Converting metric to 3DOF units
   }// for
   g->command(make3DOFCommand("SP", command));  // Send the new speed to the controller
}// velocity_command
/**
** Used to send an actuator acceleration command to
** the motion platform.
**
** @param *command a double array containing the three actuator accelerations
*/
void Model::acceleration_command(double *command){
   for(int i=0; i&lt;number_of_actuator; i++){
      command[i] *= conversion_factor; // Converting metric to 3DOF units
      command[i] = fabs(command[i]); // AC and DC must be positive only
   }// for
   make3DOFCommand("AC", command);  // Send the new acceleration to the controller
   make3DOFCommand("DC", command);  // Send the new decceleration to the controller
}// acceleration_command
/**
** Used to know if the motion platform is currently moving.
**
** @return &lt;true&gt; if the platform is moving and &lt;false&gt; if the platform is not moving
*/
bool Model::is_platform_moving(){
   bool answer = true;  // By default we will say the platform is in motion
   string platformInfo;  // String to receive the platform information
   if(platformType == "3DOF"){ // If connected to the 3DOF
      platformInfo = g->command("TV"); // Use the TV (tell velocity command) to get a reading
      if(platformInfo == "0, 0, 0"){  // If all the actuators have a speed of 0
         answer = false;  // The platform is not in motion
      }// if
   }// if
   else if (platformType == "6DOF"){  // If connected to the 6DOF
      // 6DOF CODE HERE
   }// else if
   return answer; // Return the velocity for all actuators
}// tell_platform_velocity
/**
** This method processes manual user inputs into motion platform commands.
** This method only processes events that typically
** come from user input such as a GUI. Surprisingly enough,
** it is not possible to do a switch on strings in C++. For
** this reason, if and else if has to be used to triage
** the commands.
**
** @param uInput a string containing the command request
** @return  a string containing feedback in the execution of the requested command
*/
string Model::process_user_input(string uInput){
   string feedback = "Cannot execute command while Parking!";
   if(prevent_movement == true && !is_platform_moving()){ // If the platform stopped moving, movement can resume
      prevent_movement = false;  // Set the movement prevention flag to false
      g->command("BL -5000,-5000,-5000"); // Define backward limit for all actuators (only 5000 because of platform design)
   }// if
   if(!prevent_movement){ // If there is no movement prevention
      if(platformType == "3DOF"){  // If we are connected to the 3DOF
         g->command("AB1"); // Abort any current movement of the motion platform
         if(uInput == "erasePosition"){
            g->command("BL -2147483648, -2147483648, -2147483648"); // Remove backward limits on actuators
            g->command("FL 2147483647, 2147483647, 2147483647"); // Remove forward limits on actuators
            feedback = "Parked position erased..."; // Feedback message
```

```
}// if
else if(uInput == "markPositionParked"){
   g->command("DP -6000, -6000, -6000"); // Define initial position
   g->command("FL 6000,6000,6000");  // Define forward limit for all actuators
   g->command("BL -5000,-5000,-5000");  // Define backward limit for all actuators (only 5000 because of platform design)
   feedback = "Position marked as parked..."; // Feedback message
}// else if
else if(uInput == "extend"){
   g->command("SP 2000,2000,2000");  // Set jog speed to 2000 (0.066 m/s) for all actuators
   g->command("AC 235500,235500,235500"); // Set acceleration to 235500 (7.85 m/s/s) for all actuators
   g->command("DC 235500,235500,235500"); // Set deceleration to 235500 (7.85 m/s/s) for all actuators
   g->command("PT 1,1,1"); // Register all actuator for "on the fly" absolute position move
   g->command("PA 6000,6000,6000");  // Extend the motion platform
   feedback = "Extending the platform..."; // Feedback message
}// else if
else if(uInput == "center"){
   g->command("BL -5000,-5000,-5000");  // Define backward limit for all actuators (only 5000 because of platform design)
   g->command("SP 2000,2000,2000");  // Set jog speed to 2000 (0.066 m/s) for all actuators
   g->command("AC 235500,235500,235500"); // Set acceleration to 235500 (7.85 m/s/s) for all actuators
   g->command("DC 235500,235500,235500"); // Set deceleration to 235500 (7.85 m/s/s) for all actuators
   g->command("PT 1,1,1"); // Register all actuator for "on the fly" absolute position move
   g->command("PA 0,0,0"); // Center the motion platform
   feedback = "Moving platform to central position..."; // Feedback message
}// else if
else if(uInput == "park"){
   g->command("SP 2000,2000,2000");  // Set jog speed to 2000 for all actuators
   g->command("AC 235500,235500,235500"); // Set acceleration
   g->command("DC 235500,235500,235500");  // Set deceleration
   g->command("PT 1,1,1"); // Register all actuator for "on the fly" absolute position move
   g->command("BL -6000,-6000,-6000");  // Define backward limit for all actuators
   g->command("PA -6000,-6000,-6000");  // Park the simulator
   prevent_movement = true; // When the platform is parking prevent other movements
   feedback = "Parking the platform..."; // Feedback message
}// else if
else if(uInput == "stopMovement"){
   g->command("STXYZ"); // Stops any current movement of the motion platform
   feedback = "Movement Stopped...";  // Feedback message
}// else if
else if(uInput == "pitchForward"){
   g->command("JG -" + speed + "," + speed + "," + speed); // Actuator 1 down, actuators 2 and 3 up
   g->command("BGXYZ"); // Begin motion on all actuators
   feedback = "Pitching forward...";  // Feedback message
}// else if
else if(uInput == "pitchBackward"){
   g->command("JG " + speed + ",-" + speed + ",-" + speed);// Actuator 1 up, actuators 2 and 3 down
   g->command("BGXYZ"); // Begin motion on all actuators
   feedback = "Pitching forward...";  // Feedback message
}// else if
else if(uInput == "rollLeft"){
   g->command("JG ," + speed + ",-" + speed); // Actuator 2 up and actuator 3 down
   g->command("BGYZ");  // Begin motion on actuators 2 and 3
   feedback = "Rolling left...";  // Feedback message
}// else if
else if(uInput == "rollRight"){
   g->command("JG ,-" + speed + "," + speed); // Actuator 2 down and actuator 3 up
   g->command("BGYZ");  // Begin motion on actuators 2 and 3
   feedback = "Rolling right...";  // Feedback message
}// else if
else if(uInput == "heaveUp"){
   g->command("JG " + speed + "," + speed + "," + speed); // All actuator up
   g->command("BGXYZ"); // Begin motion on all actuators
   feedback = "Heaving up..."; // Feedback message
}// else if
else if(uInput == "heaveDown"){
   g->command("JG -" + speed + ",-" + speed + ",-" + speed);// All actuator down
   g->command("BGXYZ");  // Begin motion on all actuators
   feedback = "Heaving down..."; // Feedback message
}// else if
else if(uInput == "actuator1Up"){
   g->command("JG " + speed); // Actuator 1 up
```

```
        g->command("BGX"); // Begin motion on actuator 1
        feedback = "Actuator 1 moving up...";  // Feedback message
    }// else if
    else if(uInput == "actuator1Down"){
        g->command("JG -" + speed); // Actuator 1 down
        g->command("BGX"); // Begin motion on actuator 1
        feedback = "Actuator 1 moving down...";  // Feedback message
    }// else if
    else if(uInput == "actuator2Up"){
        g->command("JG ," + speed); // Actuator 2 up
        g->command("BGY"); // Begin motion on actuator 2
        feedback = "Actuator 2 moving up...";  // Feedback message
    }// else if
    else if(uInput == "actuator2Down"){
        g->command("JG ,-" + speed); // Actuator 2 down
        g->command("BGY"); // Begin motion on actuator 2
        feedback = "Actuator 2 moving down...";  // Feedback message
    }// else if
    else if(uInput == "actuator3Up"){
        g->command("JG ,," + speed); // Actuator 3 up
        g->command("BGZ"); // Begin motion on actuator 3
        feedback = "Actuator 3 moving up...";  // Feedback message
    }// else if
    else if(uInput == "actuator3Down"){
        g->command("JG ,,-" + speed); // Actuator 3 down
        g->command("BGZ"); // Begin motion on actuator 3
        feedback = "Actuator 3 moving down...";  // Feedback message
    }// else if
}// if
/**
**
** 6DOF MOTION CODE TO BE INSERTED BELOW
** Specific 6DOF commands to be inserted for each motion type.
**
*/
else if(platformType == "6DOF"){ // If we are connected to the 6DOF
    if(uInput == "erasePosition"){
        // 6DOF CODE HERE
        feedback = "Parked position erased..."; // Feedback message
    }// if
    else if(uInput == "markPositionParked"){
        // 6DOF CODE HERE
        feedback = "Position marked as parked..."; // Feedback message
    }// else if
    else if(uInput == "center"){
        // 6DOF CODE HERE
        feedback = "Moving platform to central position..."; // Feedback message
    }// else if
    else if(uInput == "park"){
        // 6DOF CODE HERE
        feedback = "Parking the platform..."; // Feedback message
    }// else if
    else if(uInput == "stopMovement"){
        // 6DOF CODE HERE
        feedback = "Movement Stopped...";  // Feedback message
    }// else if
    else if(uInput == "pitchForward"){
        // 6DOF CODE HERE
        feedback = "Pitching forward...";  // Feedback message
    }// else if
    else if(uInput == "pitchBackward"){
        // 6DOF CODE HERE
        feedback = "Pitching backward...";  // Feedback message
    }// else if
    else if(uInput == "rollLeft"){
        // 6DOF CODE HERE
        feedback = "Rolling left...";  // Feedback message
    }// else if
    else if(uInput == "rollRight"){
        // 6DOF CODE HERE
```

```
    feedback = "Rolling right...";  // Feedback message
}// else if
else if(uInput == "heaveUp"){
    // 6DOF CODE HERE
    feedback = "Heaving up..."; // Feedback message
}// else if
else if(uInput == "heaveDown"){
    // 6DOF CODE HERE
    feedback = "Heaving down...";  // Feedback message
}// else if
else if(uInput == "yawLeft"){
    // 6DOF CODE HERE
    feedback = "Yawing left...";  // Feedback message
}// else if
else if(uInput == "yawRight"){
    // 6DOF CODE HERE
    feedback = "Yawing right...";  // Feedback message
}// else if
else if(uInput == "swayLeft"){
    // 6DOF CODE HERE
    feedback = "Swaying left...";  // Feedback message
}// else if
else if(uInput == "swayRight"){
    // 6DOF CODE HERE
    feedback = "Swaying right...";  // Feedback message
}// else if
else if(uInput == "surgeForward"){
    // 6DOF CODE HERE
    feedback = "Surging forward...";  // Feedback message
}// else if
else if(uInput == "surgeBackward"){
    // 6DOF CODE HERE
    feedback = "Surging backward...";  // Feedback message
}// else if
else if(uInput == "actuator1Up"){
    // 6DOF CODE HERE
    feedback = "Actuator 1 moving up..."; // Feedback message
}// else if
else if(uInput == "actuator1Down"){
    // 6DOF CODE HERE
    feedback = "Actuator 1 moving down..."; // Feedback message
}// else
else if(uInput == "actuator2Up"){
    // 6DOF CODE HERE
    feedback = "Actuator 2 moving up..."; // Feedback message
}// else if
else if(uInput == "actuator2Down"){
    // 6DOF CODE HERE
    feedback = "Actuator 2 moving down..."; // Feedback message
}//else if
else if(uInput == "actuator3Up"){
    // 6DOF CODE HERE
    feedback = "Actuator 3 moving up..."; // Feedback message
}// else if
else if(uInput == "actuator3Down"){
    // 6DOF CODE HERE
    feedback = "Actuator 3 moving down..."; // Feedback message
}// else if
else if(uInput == "actuator4Up"){
    // 6DOF CODE HERE
    feedback = "Actuator 4 moving up..."; // Feedback message
}// else if
else if(uInput == "actuator4Down"){
    // 6DOF CODE HERE
    feedback = "Actuator 4 moving down..."; // Feedback message
}// else if
else if(uInput == "actuator5Up"){
    // 6DOF CODE HERE
    feedback = "Actuator 5 moving up..."; // Feedback message
}// else if
```

```
        else if(uInput == "actuator5Down"){
            // 6DOF CODE HERE
            feedback = "Actuator 5 moving down..."; // Feedback message
        }// else if
        else if(uInput == "actuator6Up"){
            // 6DOF CODE HERE
            feedback = "Actuator 6 moving up..."; // Feedback message
        }// else if
        else if(uInput == "actuator6Down"){
            // 6DOF CODE HERE
            feedback = "Actuator 6 moving down..."; // Feedback message
        }// else if
      }//else if
   }// if
   return feedback;
}// process_user_input
```

Udp.h

```
/**
** Udp.h
** IDE: Visual C++ 2008 Express Edition
** @author Martin Vidovic
** @author 2Lt Jean Le Bouthillier
** Latest Revision: 13 November 2009
**
*/
/**
** Description of UDP
**
** UDP is in nature a datagram protocol. A datagram is a block of data with
** a well known size. When communicating using UDP we are sending datagrams
** and receiving datagrams. With each send() or recv() operation exactly
** one complete datagram is sent or received.
**
** With contrast to TCP, there is no connection in UDP. A program reading
** datagram packets can do so from multiple sources on the same socket
** simultaneously. This is a unique feature that can be taken advantage of
** in certain applications.
**
** However UDP has a disadvantage over TCP, since it does not offer any data
** reception acknowledgements. This means, that we may be sending datagrams
** but the other side of the communications channel will not received it.
** This usually happens on unreliable data links or if the data link is
** overloaded.
**
** To sum up, the essential thing to remember is the datagram (a packet of
** arbitrary data). When doing send()/sendto(), you are sending a datagram --
** one undividable packet of data. When doing recv()/recvfrom(), you are
** waiting for one undividable packet of data. Everything else is then
** completely dependent on the specific application protocol.
**
*/
#pragma once
#ifndef _SP_UDP_H_
#define _SP_UDP_H_
#include &lt;stdlib.h&gt;
#include &lt;string.h&gt;
#include &lt;winsock2.h&gt;
#include &lt;Ws2tcpip.h&gt;
#include &lt;windows.h&gt;
typedef unsigned short PortNumber; // Convenience typedef for Port parameter
class Udp{
   public:
   Udp();
   ~Udp();
   bool IsOk();
   private:
   /**
```

```
   ** Flag used to hold the success of Winsock Initialization
   */
   bool success;
};
bool IsWsaReady(); // Method to query Windows sockets status
#endif
```

Udp.cpp

```
/**
** Udp.cpp
** IDE: Visual C++ 2008 Express Edition
** @author Martin Vidovic
** Modified by: 2Lt Jean Le Bouthillier
** Latest Revision: 13 November 2009
**
** Description:
** A helper class to make web service adressing (WSA) Windows socket (Winsock) initialization
** and finalization easier and more reliable.
*/
#include "Udp.h"
/**
** WsaHelper Constructor
*/
Udp::Udp(){
   WSAData wsaData; // WSAData used to store windows socket initialization informations
   /**
   ** Initialize windows socket, 0 is returned if
   ** there are no errors in the initialization.
   */
   this->success = (WSAStartup(MAKEWORD(2, 2), &wsaData) == 0);
}// WsaHelper
/**
** WsaHelper Destructor
*/
Udp::~Udp(){
   if (this->success == true){ // If there is an initialized Winsock
      WSACleanup(); // Stop networking functions
      this->success = false; // Set the flag to false
   }// if
}// ~WsaHelper
/**
** This method returns the state of the winsock initialization.
**
** @return &lt;true&gt; if there is an initialized Winsock, &lt;false&gt; if there is no initialized Winsock
*/
bool Udp::IsOk(){
   return this->success;
}// IsOk
/**
** Convenience Udp instance to check for Winsock status
*/
Udp udp;
/**
** This method is used globally to check the status of the Winsock.
**
** @return &lt;true&gt; if the Winsock is ready, &lt;false&gt; otherwise
*/
bool IsWsaReady(){
   return udp.IsOk();
}// IsWsaReady
```

UdpReceive.h

```
/**
** UdpReceive.h
** IDE: Visual C++ 2008 Express Edition
** @author Martin Vidovic
```

```
** Modified by: 2Lt Jean Le Bouthillier
** Latest Revision: 13 November 2009
**
*/
#pragma once
#ifndef _SP_UDP_RECEIVE_H_
#define _SP_UDP_RECEIVE_H_
#include "Udp.h"
class UdpReceive{
   private:
   /**
   ** Socket used for datagram reception
   */
   SOCKET mSocket;
   /**
   ** Address (IP and port) of the datagram origin
   */
   sockaddr_in mSourceAddr;
   void SocketCleanup();
   bool ReceiveDatagram(void *nData, size_t nDataSize, long nTimeoutMs);
   public:
   // Class constructor.
   UdpReceive();
   // Class destructor.
   ~UdpReceive();
   bool Initialize(const char *nSourceIp, PortNumber nSourcePort);
   bool ReceiveDouble(double *nValue, long nTimeoutMs);
   bool ReceiveDoubleArray(double *nValues, size_t nNumberValues, long nTimeoutMs);
};
#endif
```

UdpReceive.cpp

```
/**
** UdpReceive.cpp
** IDE: Visual C++ 2008 Express Edition
** @author Martin Vidovic
** Modified by: 2Lt Jean Le Bouthillier
** Latest Revision: 13 November 2009
**
**
** Description:
** A helper class to make web service adressing (WSA) Windows socket (Winsock) initialization
** and finalization easier and more reliable.
*/
#include "UdpReceive.h"
/**
** UdpReceive Constructor
*/
UdpReceive::UdpReceive(){
   // Initialize the mSocket to an initial value.
   this->mSocket = INVALID_SOCKET;
}// UdpReceive
/**
** UdpReceive Destructor
*/
UdpReceive::~UdpReceive(){
   // Release networking resources.
   this->SocketCleanup();
}// ~UdpReceive
/**
** This method is used to close the current network socket.
*/
void UdpReceive::SocketCleanup(){
   // Close the socket.
   if (this->mSocket != INVALID_SOCKET){
      closesocket(this->mSocket); // Closes the socket.
      this->mSocket = INVALID_SOCKET;
   }// if
```

```
}// SocketCleanup
/**
** This method is used to create a network socket and
** initialize the UDP networking.
**
** @param *nSourceIp a pointer to the IP address of the Datagram source
** @param nSourcePort the port number of the Datagram source
** @return &lt;true&gt; if the initialization was succesful, &lt;false&gt; otherwise
*/
bool UdpReceive::Initialize(const char *nSourceIp, PortNumber nSourcePort){
   // Check for invalid state.
   if (this->mSocket != INVALID_SOCKET){
      return false;
   }// if
   if (nSourceIp == NULL){
      return false;
   }// if
   /**
   ** Creating a network socket.
   ** AF_INET - We are using IP (Internet) family of protocols.
   ** SOCK_DGRAM - We are using datagram-style communication.
   ** IPPROTO_UDP - We are using UDP protocol.
   */
   this->mSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   if (this->mSocket == INVALID_SOCKET){ // Check if socket was successfully created.
      return false;
   }// if
   /**
   ** Set socket SO_REUSEADDR option. This makes reuse of the socket possible.
   */
   int solEnable = 1;
   int setSockOptResult = setsockopt
   (
   this->mSocket, // Socket to change options on
   SOL_SOCKET,  // Level of options - We are changing options on the socket level
   SO_REUSEADDR, // The option ID
   (char *) &solEnable,  // New option value
   sizeof(solEnable) // Size of the value data type.
   );
   if (setSockOptResult != 0){  // If setting of the socket options failed
      return false;
   }// if
   /**
   ** Creating source address structure
   ** Bind our socket to the source port number and address.
   ** This is needed so that we receive only packets sent to this address.
   */
   ZeroMemory(&this->mSourceAddr, sizeof(this->mSourceAddr)); // Initialize sourceAddr stucture
   this->mSourceAddr.sin_family = AF_INET;  // IP family of address
   /**
   ** Set port to nPort, since we are sending _from_ this port.
   ** Use htons to convert from little-endian PC format to big-endian network format.
   */
   this->mSourceAddr.sin_port = htons(nSourcePort);
   /**
   ** Our socket can be bound to any IP address. The Datagram will
   ** be assigned a source IP address from the egress network interface.
   */
   this->mSourceAddr.sin_addr.s_addr = inet_addr(nSourceIp);
   /**
   ** Bind source address to our socket.
   */
   int bindResult = bind(this->mSocket, (sockaddr *) &this->mSourceAddr, sizeof(this->mSourceAddr));
   if (bindResult != 0){ // If bind() failed for some reason
      this->SocketCleanup(); // Close the current socket
      return false;  // Return failure
   }// if
   return true;
}// Initialize
/**
```

```
** This method reads one UDP datagram from the network and puts the information
** from this datagram in a data buffer at a specified memory adress.
**
** @param *nData pointer to a valid data buffer to receive datagram bytes
** @param nDataSize contains the number of bytes available in nData buffer
** @param nTimeoutMs sets the timeout of this method if no datagram is available (-1 is infinite timeout)
** @return &lt;true&gt; if nDataSize bytes were successfully read, &lt;false&gt; otherwise
*/
bool UdpReceive::ReceiveDatagram(void *nData, size_t nDataSize, long nTimeoutMs){
   fd_set readSet; // Initialize the fd_set structure used for waiting
   FD_ZERO(&readSet);  // Initializes readSet to have zero bits for all file descriptors
   FD_SET(this->mSocket, &readSet);  // Set the bit for mSocket in readSet
   timeval tval; // Setup timeout
   tval.tv_sec = nTimeoutMs / 1000;  // In seconds
   tval.tv_usec = (nTimeoutMs % 1000) * 1000; // In useconds
   int selRet = 1; // Integer to hold return of select()
   if (nTimeoutMs &gt;= 0){
      /**
      ** Wait for a datagram up to nTimeoutMs milli seconds.
      ** Select() returns 1 if datagram is waiting to be received
      ** and 0 if there is no datagram to be received.
      */
      selRet = select(this->mSocket + 1, &readSet, NULL, NULL, &tval);
   }// if
   if (selRet &gt; 0){  // Some data is ready or user wants to wait for the datagram
      int bytesRecv = recv(
      this->mSocket,  // Socket to receive from
      (char *) nData,  // User defined buffer to store received data
      nDataSize,  // Number of bytes to receive
      0 // No flags are needed
      );
      if (bytesRecv == nDataSize){  // If the expected number of bytes were read
         return true;  // Return success
      }// if
   }// if
   return false; // Return failure
}// ReceiveDatagram
/**
** This method is used to receive a double value
** over the network through a UDP datagram.
**
** @param *nValue a pointer to a double variable to store the incoming datagram
** @param nTimeoutMs the timeout in milliseconds to wait for the double value
** @return &lt;true&gt; if a double value was succesfully received, &lt;false&gt; otherwise
*/
bool UdpReceive::ReceiveDouble(double *nValue, long nTimeoutMs){
   /**
   ** Check if Winsock is ready and socket is created
   */
   if (IsWsaReady() == false || this->mSocket == INVALID_SOCKET) // Check if the object state is valid
   return false; // If something went wrong
   /**
   ** Receive the datagram
   */
   return this->ReceiveDatagram(nValue, sizeof(*nValue), nTimeoutMs);
}// ReceiveDouble
/**
** This method is used to receive a double array
** over the network through a UDP datagram.
**
** @param *nValues a pointer to a double array to store the incoming datagram
** @param nNumberValues the expected size of the double array coming over UDP
** @param nTimeoutMs the timeout in milliseconds to wait for the double array
** @return &lt;true&gt; if a double value was succesfully received, &lt;false&gt; otherwise
*/
bool UdpReceive::ReceiveDoubleArray(double *nValues, size_t nNumberValues, long nTimeoutMs){
   /**
   ** Check if Winsock is ready and socket is created
   */
   if (IsWsaReady() == false || this->mSocket == INVALID_SOCKET) // Check if the object state is valid
```

```
   return false; // If something went wrong
   /**
   ** Receive the datagram
   */
   return this->ReceiveDatagram(nValues, nNumberValues * sizeof(*nValues), nTimeoutMs);
}// ReceiveDoubleArray
```

UdpSend.h

```
/**
** UdpSend.h
** IDE: Visual C++ 2008 Express Edition
** @author Martin Vidovic
** Modified by: 2Lt Jean Le Bouthillier
** Latest Revision: 13 November 2009
**
*/
#pragma once
#ifndef _SP_UDP_SEND_H_
#define _SP_UDP_SEND_H_
#include "Udp.h"
class UdpSend{
   private:
   /**
   ** IP socket used for sending UDP Datagram. The socket is
   ** an IP and port pair and must be unique on the system.
   */
   SOCKET mSocket;
   /**
   ** A structure containing the destination address template.
   ** It is a template since the Port field is not set, only the IP address is set.
   ** The port value is set in SendDouble()/SendFloat() for each invocation.
   */
   sockaddr_in mDestAddr;
   void SocketCleanup();
   bool SendDatagram(PortNumber nDestPort, const void *nData, size_t nDataSize);
   public:
   UdpSend();
   ~UdpSend();
   bool Initialize(const char *nDestIp);
   bool SendDouble(PortNumber nPort, double nValue);
};
#endif
```

UdpSend.cpp

```
/**
** UdpSend.cpp
** IDE: Visual C++ 2008 Express Edition
** @author Martin Vidovic
** Modified by: 2Lt Jean Le Bouthillier
** Latest Revision: 13 November 2009
**
** Description:
** This class is used to send UDP datagram over the network.
** It is currently only capable of sending double values over
** UDP datagram, but it could be modified to send any kind of data
** over the network.
**
*/
#include "UdpSend.h"
/**
** UdpSend Constructor
*/
UdpSend::UdpSend(){
   // Initialize the mSocket to an initial value.
   this->mSocket = INVALID_SOCKET;
}// UdpSend
```

```
/**
** UdpSend Destructor
*/
UdpSend::~UdpSend(){
   // Release networking resources.
   this->SocketCleanup();
}// ~UdpSend
/**
** This method is used to close the current network socket.
*/
void UdpSend::SocketCleanup(){
   // Close the socket.
   if (this->mSocket != INVALID_SOCKET){
      closesocket(this->mSocket); // Closes the socket.
      this->mSocket = INVALID_SOCKET;
   }// if
}// SocketCleanup
/**
** This method is used to create a network socket and
** initialize the UDP networking.
**
** @param *nDestIp a pointer to the first character of the string containing the destination ip adress
*/
bool UdpSend::Initialize(const char *nDestIp){
   if (this->mSocket != INVALID_SOCKET){ // Check for invalid state
      return false;  // Return a failure
   }// if
   if (nDestIp == NULL){ // Make sure there is an destination ip adress
      return false;  // Return failure
   }// if
   /**
   ** Creating a network socket.
   ** AF_INET - We are using IP (Internet) family of protocols.
   ** SOCK_DGRAM - We are using datagram-style communication.
   ** IPPROTO_UDP - We are using UDP protocol.
   */
   this->mSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   if (this->mSocket == INVALID_SOCKET){ // Check if socket was successfully created
      return false;  // Return failure
   }// if
   /**
   ** Create destination address structure
   */
   ZeroMemory(&this->mDestAddr, sizeof(this->mDestAddr)); // Initialize mDestAddr stucture
   this->mDestAddr.sin_family = AF_INET; // IP family of address
   this->mDestAddr.sin_port = 0;  // Initialize destination port with 0. It will be set later in SendDatagram()
   this->mDestAddr.sin_addr.s_addr = inet_addr(nDestIp); // Convert string IP address to numeric representation
   if (this->mDestAddr.sin_addr.s_addr == INADDR_NONE){ // Check if IP string conversion was successful
      this->SocketCleanup(); // If it wasnt, close the current socket
      return false;  // Return failure
   }// if
   return true; // Everything went well, return a success
}// Initialize
/**
** This method is used to send a UDP Datagram over the network.
**
** @param nDestPort the destination port for this datagram
** @param nData a pointer to the memory block containing the data to be sent
** @param nDataSize number of bytes of data to be sent
** @return &lt;true&gt; if the the number of bytes sent matches the number of bytes intended to send, &lt;false&gt; otherwise
*/
bool UdpSend::SendDatagram(PortNumber nDestPort, const void *nData, size_t nDataSize){
   /**
   ** Prepare the destination address
   */
   sockaddr_in destAddr = this->mDestAddr; // Destination IP was set in Initialize()
   destAddr.sin_port = htons(nDestPort); // Change the destination port. htons() converts from host to network format.
   /**
   ** Send the UDP datagram to the network. The Datagram will be assigned a source
   ** IP address from the egress network interface. The source port is chosen
```

```
    ** randomly by Windows.
    */
    int sendToResult = sendto
    (
    this->mSocket,  // Send using this socket (source IP and port)
    (char *) nData,  // Send bytes from this memory block
    nDataSize,  // Number of bytes to send in this datagram
    0, // No flags are required
    (sockaddr *) &destAddr, // Send to this IP address and port (destination IP address and port)
    sizeof(destAddr) // This is just the size of destAddr structure (it's needed by Windows)
    );
    return (sendToResult == nDataSize); // Datagram sent successfully if this condition is met
}// SendDatagram
/**
** This method is used to send a double value over UDP datagram
** to a specific port number.
**
** @param nPort the port number to which the datagram will be sent
** @param nValue the double value to be sent over UDP
** @return  &lt;true&gt; if the datagram was succesfully sent, &lt;false&gt; otherwise
*/
bool UdpSend::SendDouble(PortNumber nPort, double nValue){
    /**
    ** Check if Winsock is ready and socket is created
    */
    if (IsWsaReady() == false || this->mSocket == INVALID_SOCKET){ // Check if the object state is valid
        return false; // If something went wrong
    }// if
    return this->SendDatagram(nPort, &nValue, sizeof(nValue)); // Send the datagram
}// SendDouble
```

REFERENCES

[1] Page, R.L., "Brief History of Flight Simulation", Proceedings of SimTecT 2000 Conference, Sydney, 2000

[2] CAE Website, Motion Simulators, Accessed on 23 October 2009 http://www.cae.com/en/sim.products/sim.products.asp

[3] Pouliot, N.A., Gosselin, C.M., Nahon, M.A., "Motion Simulation Capabilities of Three-Degree-of-Freedom Flight Simulators", Journal of Aircraft, Vol. 35, No. 1, January 1998, pp. 9-17

[4] Reid, L.D., Nahon, M.A., "Response of Airline Pilots to Variations in Flight Simulator Motion Algorithms", Journal of Aircraft, Vol. 25, No. 7, July 1988

[5] In Motion Simulation Website, 3DOF, Accessed on 23 October 2009 http://www.inmotionsimulation.com/3DOF.html

[6] Sarnicola Simulation Systems, Inc., Triad 1500-E Motion Platform Owner's Manual.

[7] Land Force Command Order 28-01 – Land Force Simulation Policy in Support of Land Force Training

[8] Reid, L., Nahon, M. "Flight Simulator Motion-Base Drive Algorithms: Part 1 – Developing and Testing the Equations", University of Toronto, Ontario, Canada, UTIAS Rept. 296, Dec. 1985

[9] Reid, L., Nahon, M. "Flight Simulator Motion-Base Drive Algorithms: Part 2 – Selecting the System Parameters", University of Toronto, Ontario, Canada, UTIAS Rept. 307, May 1986

[10] Reid, L., Nahon, M., "Flight Simulator Motion-Base Drive Algorithms: Part 3 – Pilot Evaluations", University of Toronto, Ontario, Canada, UTIAS Rept. 319, Dec. 1986

[11] Reid, L., Nahon, M., "Simulator Motion Drive Algorithms – A Designer's

Perspective", Journal of Guidance, Control, and Dynamics, Vol. 13, No. 2, 1990, pp. 356-362.

[12] Grant, P., Reid, L., "Motion Washout Filter Tuning: Rules and Requirements", Journal of Aircraft, Vol. 34, No.2, March-April 1997, pp. 145-151

[13] Haik, Y., "Engineering Design Process", Brooks/Cole - Thomson Learning, 2003, 270 p.

[14] Burki-Cohen, J., Soja, N., Longridge, T., "Simulator Fidelity Requirements: The Case of Platform Motion", 9th ITEC International Training & Education Conference, Lausanne, Switzerland, pp. 216-231

[15] Mooney, J.D., "Bringing Portability to the Software Process", Dept. of Statistics and Computer Science, West Virginia University, Morgantown WV.

[16] IEEE Std. 610.12-1990, "Glossary of Software Engineering Terminology" in Software Engineering Standards Collection, IEEE CS Press, Los Alamitos, California, 1993

[17] Fowler, M., "UML Distilled Third Edition - A Brief Guide to the Standard Object Modelling Language", Addison-Wesley, Aug. 2009

[18] Weisfeld, M., "The Object-Oriented Thought Process - Third Edition", Addison-Wesley, Oct. 2009

[19] Freeman, E., Freeman, E., "Head First Design Patterns", O'Reilly, Oct. 2004

[20] About SimConnect, Microsoft ESP Developer Center, Accessed on 10 May 2010 http://msdn.microsoft.com/en-us/esp/cc835021.aspx

[21] SimConnect SDK Reference, MSDN Library, Accessed on 10 May 2010 http://msdn.microsoft.com/en-us/library/cc526983.aspx

[22] Simulink - Simulation and Model-Based Design, Accessed on 11 May 2010 http://www.mathworks.com/products/simulink/

[23] MATLAB - The Language of Technical Computing, Accessed on 11 May 2010 http://www.mathworks.com/products/matlab/

[24] Meyers, G., "Software Reliability Principles and Practice, Chapter 10: "Testing Principles", A Wiley Interscience Publication, 1976, pp. 169-195

[25] Montgomery, D.C., "Design and Analysis of Experiments - 6th Edition", John Wiley & Sons, Arizona State University, 2005

[26] Crossbow Technology, Inc., "320 Series User's Manual", Document 7430-0292-01, Revision B, September 2009

[27] Cessna Website, C-172 Skyhawk, Accessed on 30 July 2010 http://www.cessna.com/single-engine/skyhawk.html

[28] GlobalSecurity.org, T-41A/C Mescalero Article, Accessed on 30 July 2010 http://www.globalsecurity.org/military/systems/aircraft/t-41.htm

[29] Lamoureux, L. "Cinématique et dynamique des solides", Traité des Nouvelles Technologies - série Mécanique, Editions Hermès, Paris, 1992

[30] Dr. T. O'Haver - Smoothing Algorithms Website, Accessed on September 21 2010 http://terpconnect.umd.edu/~toh/spectrum/Smoothing.html#algorithms

[31] Thong-Shing Hwang, Ming-Sung Kuo, "Optimal Genetic and Adaptive Fuzzy Washout Filter Design in the Motion-Cueing Simulator", Asian Journal of Control, Vol. 10, January 2008, pp.88-95

[32] Gutridge C. J., "Three Degree-of-Freedom Simulator Motion Cueing Using Classical Washout Filters and Acceleration Feedback", Master of Science Thesis, Virginia Polytechnic Institute & State University, 2004.

[33] Bohemia Interactive Website, Virtual Battlespace 2 Webpage, Accessed on October 01 2010 http://www.bisimulations.com/